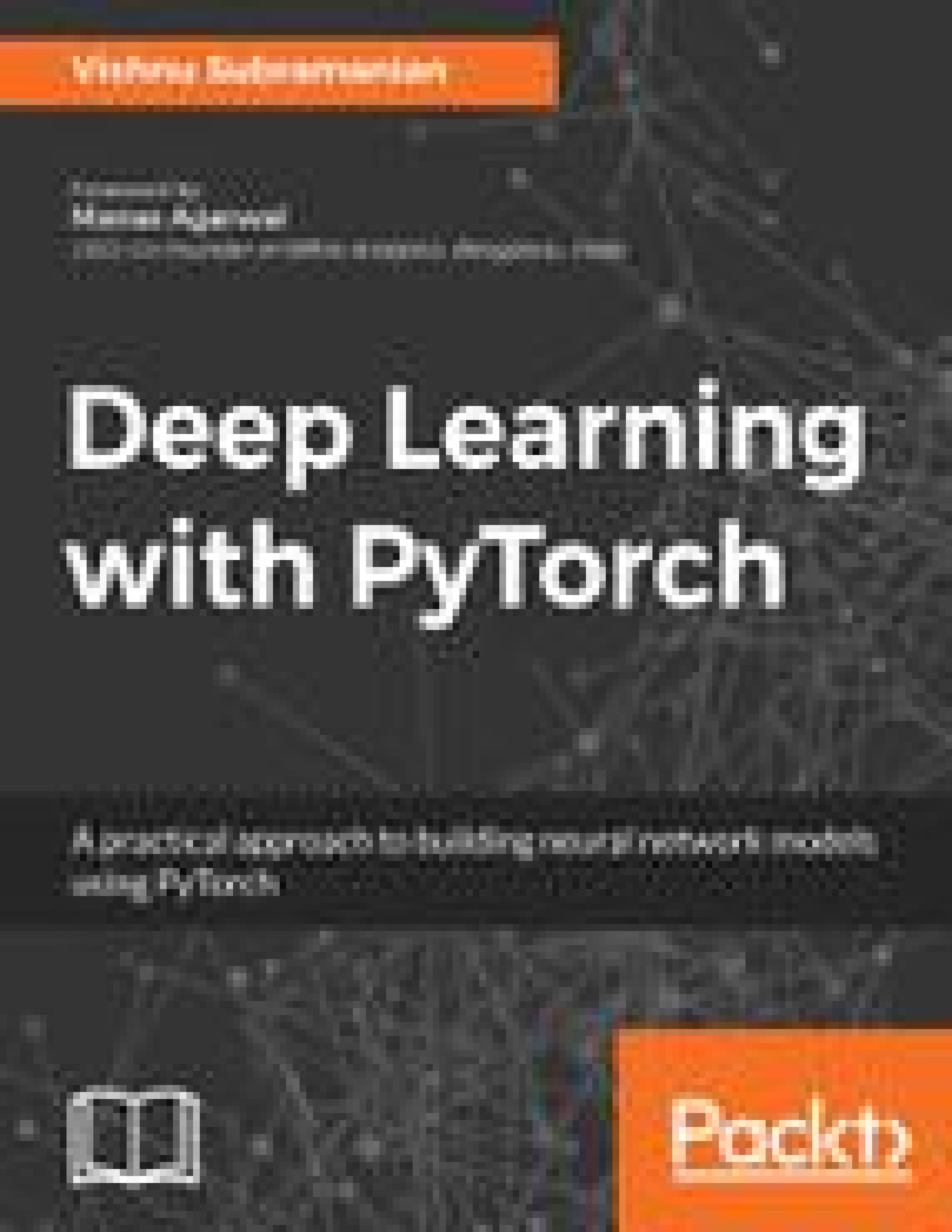


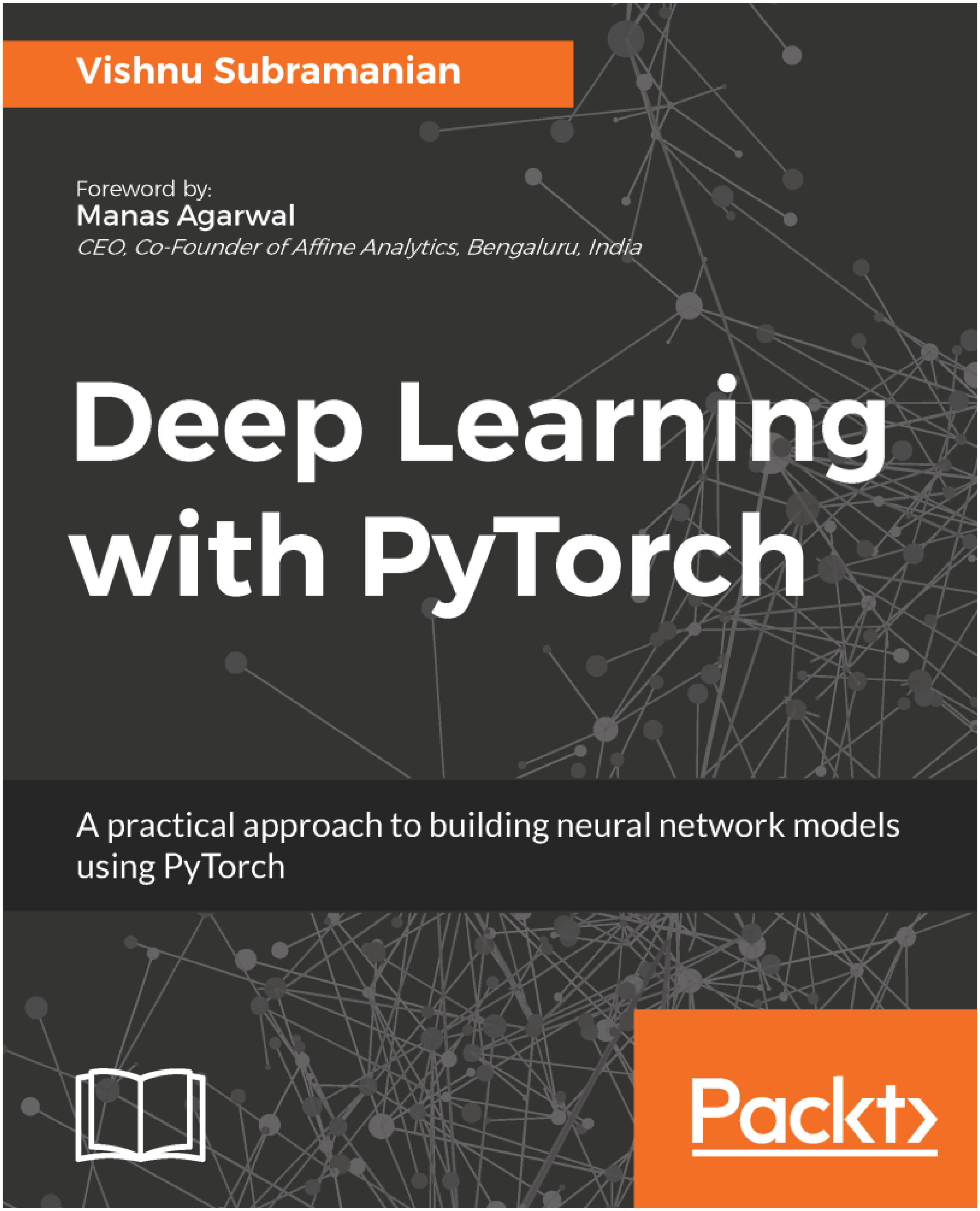
Vishnu Subramanian

Foreword by
Manas Agarwal
CEO Co-Founder of Affine Analytics, Bengaluru, India

Deep Learning with PyTorch

A practical approach to building neural network models using PyTorch

Packt



Vishnu Subramanian

Foreword by:
Manas Agarwal
CEO, Co-Founder of Affine Analytics, Bengaluru, India

Deep Learning with PyTorch

A practical approach to building neural network models using PyTorch

Packt

Deep Learning with PyTorch

A practical approach to building neural network models using PyTorch

Vishnu Subramanian

Packt

BIRMINGHAM - MUMBAI

Deep Learning with PyTorch

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Veena Pagare
Acquisition Editor: Aman Singh
Content Development Editor: Snehal Kolte
Technical Editor: Sayli Nikalje
Copy Editor: Safis Editing
Project Coordinator: Manthan Patel
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Graphics: Tania Dutta
Production Coordinator: Deepika Naik

First published: February 2018

Production reference: 1210218

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78862-433-6

`www.packtpub.com`

To Jeremy Howard and Rachel Thomas for inspiring me to write this book,
and to my family for their love.

– Vishnu Subramanian

Mapt

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Foreword

I have been working with Vishnu Subramanian for the last few years. Vishnu comes across as a passionate techno-analytical expert who has the rigor one requires to achieve excellence. His points of view on big data/machine learning/AI are well informed and carry his own analysis and appreciation of the landscape of problems and solutions. Having known him closely, I'm glad to be writing this foreword in my capacity as the CEO of Affine.

Increased success through deep learning solutions for our Fortune 500 clients clearly necessitates quick prototyping. PyTorch (a year-old deep learning framework) allows rapid prototyping for analytical projects without worrying too much about the complexity of the framework. This leads to an augmentation of the best of human capabilities with frameworks that can help deliver solutions faster. As an entrepreneur delivering advanced analytical solutions, building this capability in my teams happens to be the primary objective for me. In this book, Vishnu takes you through the fundamentals of building deep learning solutions using PyTorch while helping you build a mindset geared towards modern deep learning techniques.

The first half of the book introduces several fundamental building blocks of deep learning and PyTorch. It also covers key concepts such as overfitting, underfitting, and techniques that helps us deal with them.

In the second half of the book, Vishnu covers advanced concepts such as CNN, RNN, and LSTM transfer learning using pre-convoluted features, and one-dimensional convolutions, along with real-world examples of how these techniques can be applied. The last two chapters introduce you to modern deep learning architectures such as Inception, ResNet, DenseNet model and ensembling, and generative networks such as style transfer, GAN, and language modeling.

With all the practical examples covered and with solid explanations, this is one of the best books for readers who want to become proficient in deep learning. The rate at which technology evolves is unparalleled today. To a reader looking forward towards developing mature deep learning solutions, I would like to point that the right framework also drives the right mindset.

To all those reading through this book, happy exploring new horizons!

Wishing Vishnu and this book a roaring success, which they both deserve.

Manas Agarwal

CEO, Co-Founder of Affine Analytics,

Bengaluru, India

Contributors

About the author

Vishnu Subramanian has experience in leading, architecting, and implementing several big data analytical projects (artificial intelligence, machine learning, and deep learning). He specializes in machine learning, deep learning, distributed machine learning, and visualization. He has experience in retail, finance, and travel. He is good at understanding and coordinating between businesses, AI, and engineering teams.

This book would not have been possible without the inspiration and MOOC by Jeremy Howard and Rachel Thomas of fast.ai. Thanks to them for the important role they are playing in democratizing AI/deep learning.

About the reviewer

Poonam Ligade is a freelancer who specializes in big data tools such as Spark, Flink, and Cassandra, as well as scalable machine learning and deep learning. She is also a top kaggle kernel writer.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Preface

PyTorch is grabbing the attention of data science professionals and deep learning practitioners due to its flexibility and ease of use. This book introduces the fundamental building blocks of deep learning and PyTorch. It demonstrates how to solve real-world problems using a practical approach. You will also learn some of the modern architectures and techniques that are used to crack some cutting-edge research problems.

This book provides the intuition behind various state-of-the-art deep learning architectures, such as ResNet, DenseNet, Inception, and Seq2Seq, without diving deep into the math. It also shows how to do transfer learning, how to speed up transfer learning using pre-computed features, and how to do text classification using embeddings, pretrained embeddings, LSTM, and one-dimensional convolutions.

By the end of the book, you will be a proficient deep learning practitioner who will be able to solve some business problems using the different techniques learned here.

Who this book is for

This book is for engineers, data analysts, and data scientists, interested in deep learning, and those looking to explore and implement advanced algorithms with PyTorch. Knowledge of machine learning is helpful but not mandatory. Knowledge of Python programming is expected.

What this book covers

`Chapter 1`, *Getting Started with Deep Learning Using PyTorch*, goes over the history of **artificial intelligence** (**AI**) and machine learning and looks at the recent growth of deep learning. We will also cover how various improvements in hardware and algorithms triggered huge success in the implementation of deep learning across different applications. Finally, we will introduce the beautiful PyTorch Python library, built on top of Torch by Facebook.

`Chapter 2`, *Building Blocks of Neural Networks*, discusses the knowledge of various building blocks of PyTorch, such as variables, tensors, and `nn.module`, and how they are used to develop neural networks.

`Chapter 3`, *Diving Deep into Neural Networks*, covers the different processes involved in training a neural network, such as the data preparation, data loaders for batching tensors, the `torch.nn` package for creating network architectures and the use of PyTorch loss functions and optimizers.

`Chapter 4`, *Fundamentals of Machine Learning*, covers different types of machine learning problems, along with challenges such as overfitting and underfitting. We also cover different techniques such as data augmentation, adding dropouts, and using batch normalization to prevent overfitting.

`Chapter 5`, *Deep Learning for Computer Vision*, explains the building blocks of **Convolutional Neural Networks** (**CNNs**), such as one-dimensional and two-dimensional convolutions, max pooling, average pooling, basic CNN architectures, transfer learning, and using pre-convoluted features to train faster.

`Chapter 6`, *Deep Learning with Sequence Data and Text*, covers word embeddings, how to use pretrained embeddings, RNN, LSTM, and one-dimensional convolutions for text classification on the `IMDB` dataset.

`Chapter 7`, *Generative Networks*, explains how to use deep learning to generate artistic images, new images with DCGAN, and text using language modeling.

`Chapter 8`, *Modern Network Architectures*, explores architectures such as ResNet, Inception, and DenseNet that power modern computer vision applications. We will have a quick introduction to encoder-decoder architectures that power modern systems such as language translations and image captioning.

`Chapter 9`, *What Next?*, looks into the summarizes what we have learned and looks at keeping yourself updated in the field of deep learning.

To get the most out of this book

All the chapters (except `Chapter 1`, *Getting Started with Deep Learning Using PyTorch* and `Chapter 9`, *What Next*) have associated Jupyter Notebooks in the book's GitHub repository. The imports required for the code to run may not be included in the text to save space. You should be able to run all of the code from the Notebooks.

The book focuses on practical illustrations, so run the Jupyter Notebooks as you read the chapters.

Access to a computer with a GPU will help run the code quickly. There are companies such as `paperspace.com` and `www.crestle.com` that abstract a lot of the complexity required to run deep learning algorithms.

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Deep-Learning-with-PyTorch`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/DeepLearningwithPyTorch_ColorImages.pdf`

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The custom class has to implement two main functions, namely `__len__(self)` and `__getitem__(self, idx)`."

A block of code is set as follows:

```
x,y = get_data() # x - represents training data,y -                represent

w,b = get_weights() # w,b - Learnable parameters

for i in range(500):
    y_pred = simple_network(x) # function which computes wx + b
    loss = loss_fn(y,y_pred) # calculates sum of the squared differences of y

if i % 50 == 0:
        print(loss)
    optimize(learning_rate) # Adjust w,b to minimize the loss
```

Any command-line input or output is written as follows:

```
conda install pytorch torchvision cuda80 -c soumith
```

Bold: Indicates a new term, an important word, or words that you see onscreen.

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

Getting Started with Deep Learning Using PyTorch

Deep learning (**DL**) has revolutionized industry after industry. It was once famously described by Andrew Ng on Twitter:

Artificial Intelligence is the new electricity!

Electricity transformed countless industries; **artificial intelligence** (**AI**) will now do the same.

AI and DL are used like synonyms, but there are substantial differences between the two. Let's demystify the terminology used in the industry so that you, as a practitioner, will be able to differentiate between signal and noise.

In this chapter, we will cover the following different parts of AI:

- AI itself and its origination
- Machine learning in the real world
- Applications of deep learning
- Why deep learning now?
- Deep learning framework: PyTorch

Artificial intelligence

Countless articles discussing AI are published every day. The trend has increased in the last two years. There are several definitions of AI floating around the web, my favorite being *the automation of intellectual tasks normally performed by humans*.

The history of AI

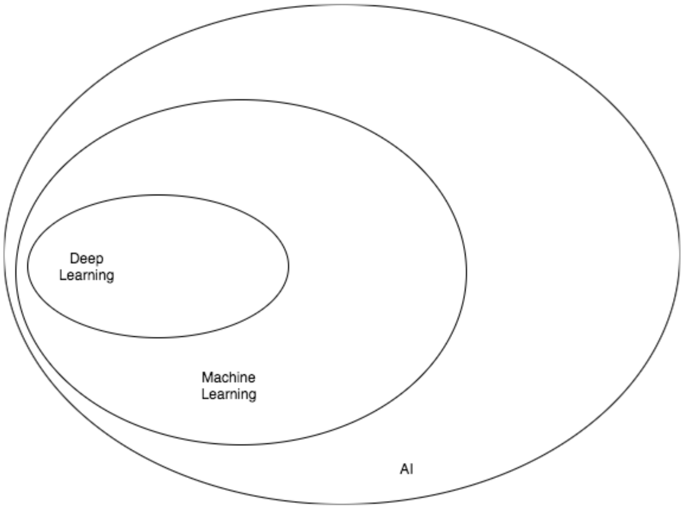
The term *artificial intelligence* was first coined by John McCarthy in 1956, when he held the first academic conference on the subject. The journey of the question of whether machines think or not started much earlier than that. In the early days of AI, machines were able to solve problems that were difficult for humans to solve.

For example, the Enigma machine was built at the end of World War II to be used in military communications. Alan Turing built an AI system that helped to crack the Enigma code. Cracking the Enigma code was a very challenging task for a human, and it could take weeks for an analyst to do. The AI machine was able to crack the code in hours.

Computers have a tough time solving problems that are intuitive to us, such as differentiating between dogs and cats, telling whether your friend is angry at you for arriving late at a party (emotions), differentiating between a truck and a car, taking notes during a seminar (speech recognition), or converting notes to another language for your friend who does not understand your language (for example, French to English). Most of these tasks are intuitive to us, but we were unable to program or hard code a computer to do these kinds of tasks. Most of the intelligence in early AI machines was hard coded, such as a computer program playing chess.

In the early years of AI, a lot of researchers believed that AI could be achieved by hard coding rules. This kind of AI is called **symbolic AI** and was useful in solving well-defined, logical problems, but it was almost incapable of solving complex problems such as image recognition, object detection, object segmentation, language translation, and natural-language-understanding tasks. Newer approaches to AI, such as machine learning and DL, were developed to solve these kinds of problems.

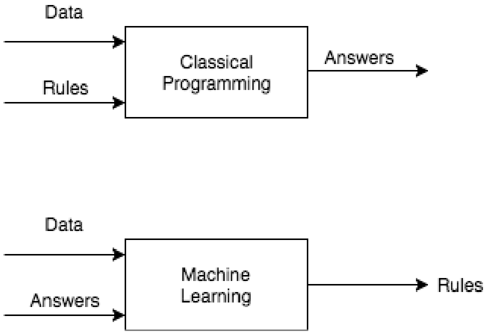
To better understand the relationships among AI, ML, and DL, let's visualize them as concentric circles with AI—the idea that came first (the largest), then machine learning—(which blossomed later), and finally DL—which is driving today's AI explosion (fitting inside both):



How AI, machine learning, and DL fit together

Machine learning

Machine learning (**ML**) is a sub-field of AI and has become popular in the last 10 years and, at times, the two are used interchangeably. AI has a lot of other sub-fields aside from machine learning. ML systems are built by showing lots of examples, unlike symbolic AI, where we hard code rules to build the system. At a high level, machine learning systems look at tons of data and come up with rules to predict outcomes for unseen data:



Machine learning versus traditional programming

Most ML algorithms perform well on structured data, such as sales predictions, recommendation systems, and marketing personalization. An important factor for any ML algorithm is feature engineering and data scientists need to spend a lot of time to get the features right for ML algorithms to perform. In certain domains, such as computer vision and **natural language processing** (**NLP**), feature engineering is challenging as they suffer from high dimensionality.

Until recently, problems like this were challenging for organizations to solve using typical machine-learning techniques, such as linear regression, random forest, and so on, for reasons such as feature engineering and high dimensionality. Consider an image of size 224 x 224 x 3 (height x width x channels), where *3* in the image size represents values of red, green, and blue color channels in a color image. To store this image in computer memory, our matrix will contain 150,528 dimensions for a single image. Assume you want to build a classifier on top of 1,000 images of size 224 x 224 x 3, the dimensions will become 1,000 times 150,528. A special branch of machine learning called **deep learning** allows you to handle these problems using modern techniques and hardware.

Examples of machine learning in real life

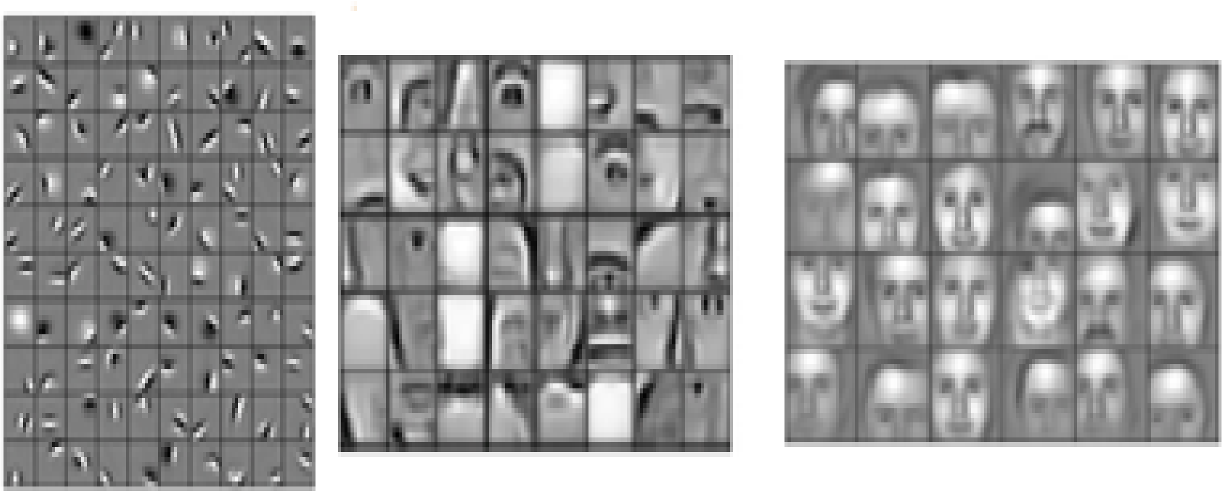
The following are some cool products that are powered by machine learning:

- **Example 1**: Google Photos uses a specific form of machine learning called **deep learning for grouping photos**
- **Example 2**: Recommendation systems, which are a family of ML algorithms, are used for recommending movies, music, and products by major companies such as Netflix, Amazon, and iTunes

Deep learning

Traditional ML algorithms use handwritten feature extraction to train algorithms, while DL algorithms use modern techniques to extract these features in an automatic fashion.

For example, a DL algorithm predicting whether an image contains a face or not extracts features such as the first layer detecting edges, the second layer detecting shapes such as noses and eyes, and the final layer detecting face shapes or more complex structures. Each layer trains based on the previous layer's representation of the data. It's OK if you find this explanation hard to understand, the later chapters of the book will help you to intuitively build and inspect such networks:



Visualizing the output of intermediate layers (Image source: https://www.cs.princeton.edu/~rajeshr/papers/cacm2011-researchHighlights-convDBN.pdf)

The use of DL has grown tremendously in the last few years with the rise of GPUs, big data, cloud providers such as **Amazon Web**

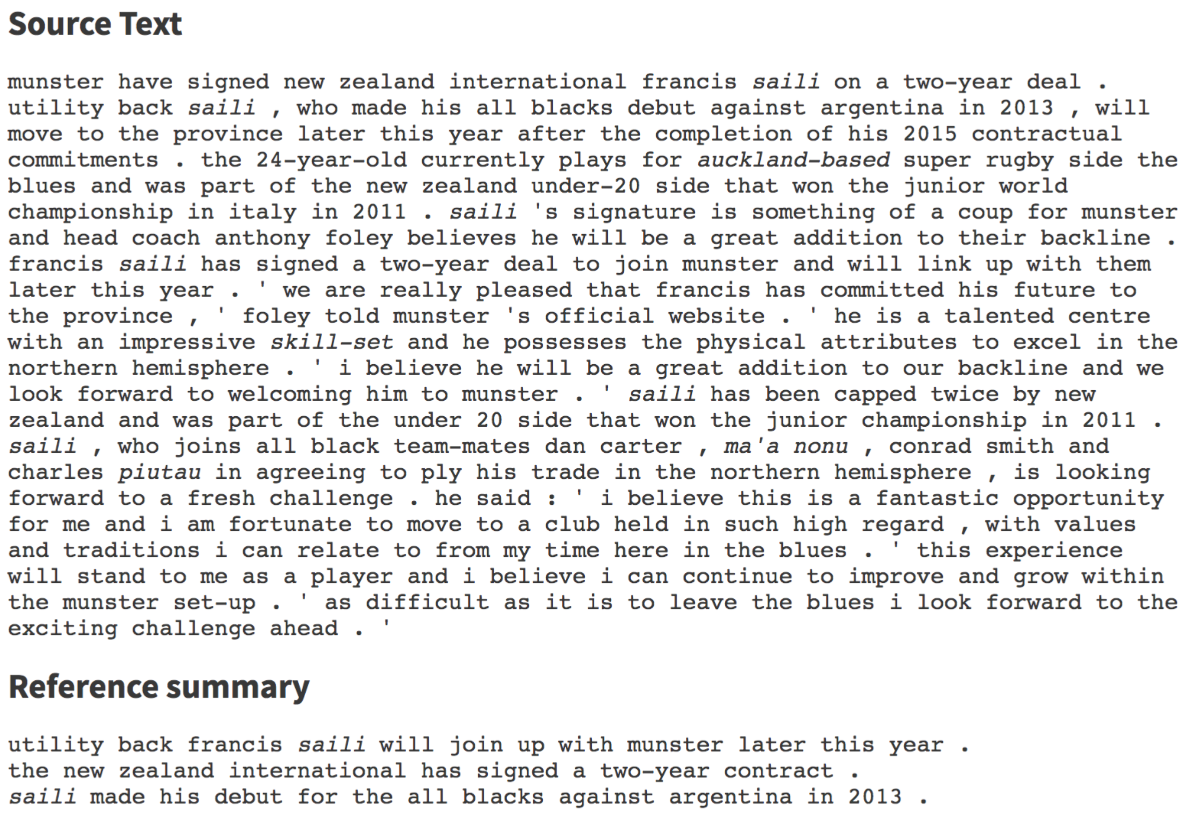
Services (**AWS**) and Google Cloud, and frameworks such as Torch, TensorFlow, Caffe, and PyTorch. In addition to this, large companies share algorithms trained on huge datasets, thus helping startups to build state-of-the-art systems on several use cases with little effort.

Applications of deep learning

Some popular applications that were made possible using DL are as follows:

- Near-human-level image classification
- Near-human-level speech recognition
- Machine translation
- Autonomous cars
- Siri, Google Voice, and Alexa have become more accurate in recent years
- A Japanese farmer sorting cucumbers
- Lung cancer detection
- Language translation beating human-level accuracy

The following screenshot shows a short example of summarization, where the computer takes a large paragraph of text and summarizes it in a few lines:



Source Text

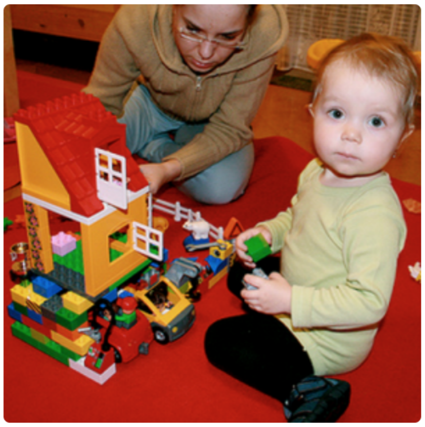
munster have signed new zealand international francis *saili* on a two-year deal . utility back *saili* , who made his all blacks debut against argentina in 2013 , will move to the province later this year after the completion of his 2015 contractual commitments . the 24-year-old currently plays for *auckland-based* super rugby side the blues and was part of the new zealand under-20 side that won the junior world championship in italy in 2011 . *saili* 's signature is something of a coup for munster and head coach anthony foley believes he will be a great addition to their backline . francis *saili* has signed a two-year deal to join munster and will link up with them later this year . ' we are really pleased that francis has committed his future to the province , ' foley told munster 's official website . ' he is a talented centre with an impressive *skill-set* and he possesses the physical attributes to excel in the northern hemisphere . ' i believe he will be a great addition to our backline and we look forward to welcoming him to munster . ' *saili* has been capped twice by new zealand and was part of the under 20 side that won the junior championship in 2011 . *saili* , who joins all black team-mates dan carter , *ma'a nonu* , conrad smith and charles *piutau* in agreeing to ply his trade in the northern hemisphere , is looking forward to a fresh challenge . he said : ' i believe this is a fantastic opportunity for me and i am fortunate to move to a club held in such high regard , with values and traditions i can relate to from my time here in the blues . ' this experience will stand to me as a player and i believe i can continue to improve and grow within the munster set-up . ' as difficult as it is to leave the blues i look forward to the exciting challenge ahead . '

Reference summary

utility back francis *saili* will join up with munster later this year .
the new zealand international has signed a two-year contract .
saili made his debut for the all blacks against argentina in 2013 .

Summary of a sample paragraph generated by computer

In the following image, a computer has been given a plain image without being told what it shows and, using object detection and some help from a dictionary, you get back an image caption stating **two young girls are playing with lego toy**. Isn't it brilliant?



"two young girls are playing with lego toy."

Object detection and image captioning (Image source: https://cs.stanford.edu/people/karpathy/cvpr2015.pdf)

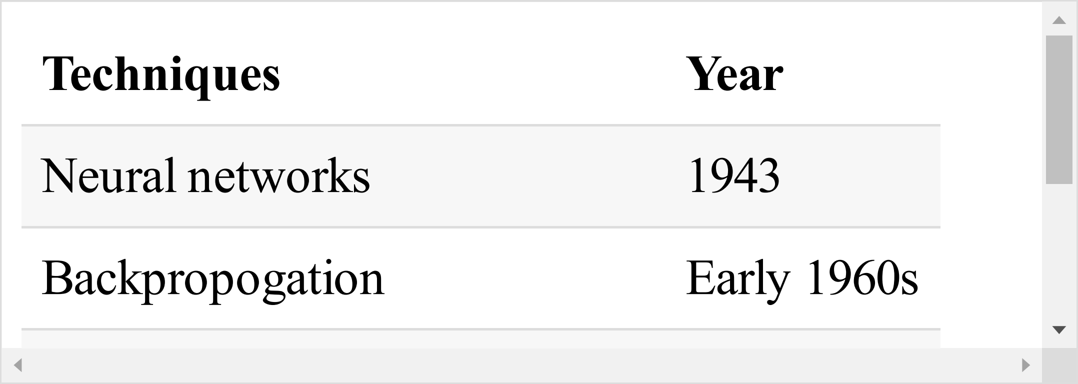
Hype associated with deep learning

People in the media and those outside the field of AI, or people who are not real practitioners of AI and DL, have been suggesting that things like the story line of the film *Terminator 2: Judgement Day* could become reality as AI/DL advances. Some of them even talk about a time in which we will become controlled by robots, where robots decide what is good for humanity. At present, the ability of AI is exaggerated far beyond its true capabilities. Currently, most DL systems are deployed in a very controlled environment and are given a limited decision boundary.

My guess is that when these systems can learn to make intelligent decisions, rather than merely completing pattern matching and, when hundreds or thousands of DL algorithms can work together, then maybe we can expect to see robots that could probably behave like the ones we see in science fiction movies. In reality, we are no closer to general artificial intelligence, where machines can do anything without being told to do so. The current state of DL is more about finding patterns from existing data to predict future outcomes. As DL practitioners, we need to differentiate between signal and noise.

The history of deep learning

Though deep learning has become popular in recent years, the theory behind deep learning has been evolving since the 1950s. The following table shows some of the most popular techniques used today in DL applications and their approximate timeline:



Techniques	Year
Neural networks	1943
Backpropogation	Early 1960s

Deep learning has been given several names over the years. It was called **cybernetics** in the 1970s, connectionism in the 1980s, and now it is either known as *deep learning* or *neural networks*. We will use DL and neural networks interchangeably. Neural networks are often referred to as an algorithms inspired by the working of human brains. However, as practitioners of DL, we need to understand that it is majorly inspired and backed by strong theories in math (linear algebra and calculus), statistics (probability), and software engineering.

Why now?

Why has DL became so popular now? Some of the crucial reasons are as follows:

- Hardware availability
- Data and algorithms
- Deep learning frameworks

Hardware availability

Deep learning requires complex mathematical operations to be performed on millions, sometimes billions, of parameters. Existing CPUs take a long time to perform these kinds of operations, although this has improved over the last several years. A new kind of hardware called a **graphics processing unit** (**GPU**) has completed these huge mathematical operations, such as matrix multiplications, orders of magnitude faster.

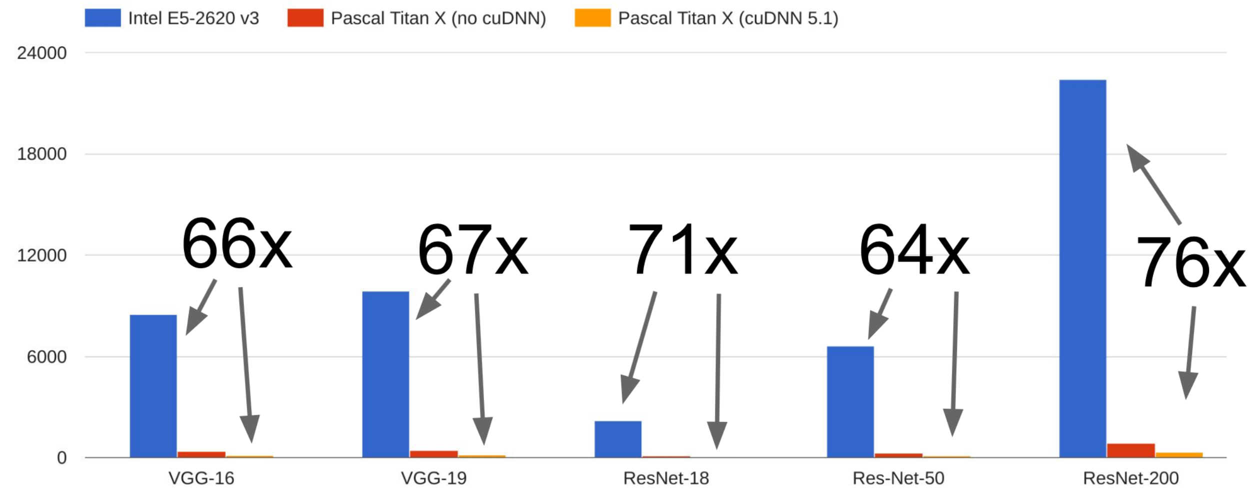
GPUs were initially built for the gaming industry by companies such as Nvidia and AMD. It turned out that this hardware is extremely efficient, not only for rendering high quality video games, but also to speed up the DL algorithms. One recent GPU from Nvidia, the *1080ti*, takes a few days to build an image-classification system on top of an `ImageNet` dataset, which previously could have taken around a month.

If you are planning to buy hardware for running deep learning, I would recommend choosing a GPU from Nvidia based on your budget. Choose one with a good amount of memory. Remember, your computer memory and GPU memory are two different things. The 1080ti comes with 11 GB of memory and it costs around $700.

You can also use various cloud providers such as AWS, Google Cloud, or Floyd (this company offers GPU machines optimized for DL). Using a cloud provider is economical if you are just starting with DL or if you are setting up machines for organization usage where you may have more financial freedom.

Performance could vary if these systems are optimized.

The following image shows some of the benchmarks that compare performance between CPUs and GPUs :



Performance benchmark of neural architectures on CPUs and GPUs (Image source: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture8.pdf)

Data and algorithms

Data is the most important ingredient for the success of deep learning. Due to the wide adoption of the internet and the growing use of smartphones, several companies, such as Facebook and Google, have been able to collect a lot of data in various formats, particularly text, images, videos, and audio. In the field of computer vision, ImageNet competitions have played a huge role in providing datasets of 1.4 million images in 1,000 categories.

These categories are hand-annotated and every year hundreds of teams compete. Some of the algorithms that were successful in the competition are VGG, ResNet, Inception, DenseNet, and many more. These algorithms are used today in industries to solve various computer vision problems. Some of the other popular datasets that are often used in the deep learning space to benchmark various algorithms are as follows:

- MNIST
- COCO dataset
- CIFAR
- The Street View House Numbers
- PASCAL VOC
- Wikipedia dump
- 20 Newsgroups

- Penn Treebank
- Kaggle

The growth of different algorithms such as batch normalization, activation functions, skip connections, **Long Short-Term Memory** (**LSTM**), dropouts, and many more have made it possible in recent years to train very deep networks faster and more successfully. In the coming chapters of this book, we will get into the details of each technique and how they help in building better models.

Deep learning frameworks

In the earlier days, people needed to have expertise in C++ and CUDA to implement DL algorithms. With a lot of organizations now open sourcing their deep learning frameworks, people with knowledge of a scripting language, such as Python, can start building and using DL algorithms. Some of the popular deep learning frameworks used today in the industry are TensorFlow, Caffe2, Keras, Theano, PyTorch, Chainer, DyNet, MXNet, and CNTK.

The adoption of deep learning would not have been this huge if it had not been for these frameworks. They abstract away a lot of underlying complications and allow us to focus on the applications. We are still in the early days of DL where, with a lot of research, breakthroughs are happening every day across companies and organizations. As a result of this, various frameworks have their own pros and cons.

PyTorch

PyTorch, and most of the other deep learning frameworks, can be used for two different things:

- Replacing NumPy-like operations with GPU-accelerated operations
- Building deep neural networks

What makes PyTorch increasingly popular is its ease of use and simplicity. Unlike most other popular deep learning frameworks, which use static computation graphs, PyTorch uses dynamic computation, which allows greater flexibility in building complex architectures.

PyTorch extensively uses Python concepts, such as classes, structures, and conditional loops, allowing us to build DL algorithms in a pure object-oriented fashion. Most of the other popular frameworks bring their own programming style, sometimes making it complex to write new algorithms and it does not support intuitive debugging. In the later chapters, we will discuss computation graphs in detail.

Though PyTorch was released recently and is still in its beta version, it has become immensely popular among data scientists and deep learning researchers for its ease of use, better performance, easier-to-debug nature, and strong growing support from various companies such as SalesForce.

As PyTorch was primarily built for research, it is not recommended for production usage in certain scenarios where latency requirements are very high. However, this is changing with a new project called **Open Neural Network Exchange** (**ONNX**) (`https://onnx.ai/`), which focuses on deploying a model developed on PyTorch to a platform like Caffe2 that is production-ready. At the time of writing, it is too early to say much about this project as it has only just been launched. The project is backed by Facebook and Microsoft.

Throughout the rest of the book, we will learn about the various Lego blocks (smaller concepts or techniques) for building powerful DL applications in the areas of computer vision and NLP.

Summary

In this introductory chapter, we explored what artificial intelligence, machine learning, and deep learning are and we discussed the differences between all the three. We also looked at applications powered by them in our day-to-day lives. We dig deeper into why DL is only now becoming more popular. Finally, we gave a gentle introduction to PyTorch, which is a deep learning framework.

In the next chapter, we will train our first neural network in PyTorch.

Building Blocks of Neural Networks

Understanding the basic building blocks of a neural network, such as tensors, tensor operations, and gradient descents, is important for building complex neural networks. In this chapter, we will build our first `Hello world` program in neural networks by covering the following topics:

- Installing PyTorch
- Implementing our first neural network
- Splitting the neural network into functional blocks
- Walking through each fundamental block covering tensors, variables, autograds, gradients, and optimizers
- Loading data using PyTorch

Installing PyTorch

PyTorch is available as a Python package and you can either use `pip`, or `conda`, to build it or you can build it from source. The recommended approach for this book is to use the Anaconda Python 3 distribution. To install Anaconda, please refer to the Anaconda official documentation at `https://conda.io/docs/user-guide/install/index.html`. All the examples will be available as Jupyter Notebooks in the book's GitHub repository. I would strongly recommend you use Jupyter Notebook, since it allows you to experiment interactively. If you already have Anaconda Python installed, then you can proceed with the following steps for PyTorch installation.

For GPU-based installation with Cuda 8:

```
conda install pytorch torchvision cuda80 -c soumith
```

For GPU-based installation with Cuda 7.5:

```
conda install pytorch torchvision -c soumith
```

For non-GPU-based installation:

```
conda install pytorch torchvision -c soumith
```

At the time of writing, PyTorch does not work on a Windows machine, so you can try a **virtual machine** (**VM**) or Docker image.

Our first neural network

We present our first neural network, which learns how to map training examples (input array) to targets (output array). Let's assume that we work for one of the largest online companies, **Wondermovies,** which serves videos on demand. Our training dataset contains a feature that represents the average hours spent by users watching movies on the platform and we would like to predict how much time each user would spend on the platform in the coming week. It's just an imaginary use case, don't think too much about it. Some of the high-level activities for building such a solution are as follows:

- **Data preparation**: The `get_data` function prepares the tensors (arrays) containing input and output data
- **Creating learnable parameters**: The `get_weights` function provides us with tensors containing random values that we will optimize to solve our problem
- **Network model**: The `simple_network` function produces the output for the input data, applying a linear rule, multiplying weights with input data, and adding the bias term ($y = Wx+b$)
- **Loss**: The `loss_fn` function provides information about how good the model is

- **Optimizer**: The `optimize` function helps us in adjusting random weights created initially to help the model calculate target values more accurately

If you are new to machine learning, do not worry, as we will understand exactly what each function does by the end of the chapter. The following functions abstract away PyTorch code to make it easier for us to understand. We will dive deep into each of these functionalities in detail. The aforementioned high level activities are common for most machine learning and deep learning problems. Later chapters in the book discuss techniques that can be used to improve each function to build useful applications.

Lets consider following linear regression equation for our neural network:

$$y = wx + b$$

Let's write our first neural network in PyTorch:

```
x,y = get_data() # x - represents training data,y -                 represent

w,b = get_weights() # w,b - Learnable parameters

for i in range(500):
    y_pred = simple_network(x) # function which computes wx + b
    loss = loss_fn(y,y_pred) # calculates sum of the squared differences of y

if i % 50 == 0:
        print(loss)
    optimize(learning_rate) # Adjust w,b to minimize the loss
```

By the end of this chapter, you will have an idea of what is happening inside each function.

Data preparation

PyTorch provides two kinds of data abstractions called `tensors` and `variables`. Tensors are similar to `numpy` arrays and they can also be used on GPUs, which provide increased performance. They provide easy methods of switching between GPUs and CPUs. For certain operations, we can notice a boost in performance and machine learning algorithms can understand different forms of data, only when represented as tensors of numbers. Tensors are like Python arrays and can change in size. For example, images can be represented as three-dimensional arrays (height, weight, channel (RGB)). It is common in deep learning to use tensors of sizes up to five dimensions. Some of the commonly used tensors are as follows:

- Scalar (0-D tensors)
- Vector (1-D tensors)
- Matrix (2-D tensors)
- 3-D tensors
- Slicing tensors
- 4-D tensors
- 5-D tensors
- Tensors on GPU

Scalar (0-D tensors)

A tensor containing only one element is called a **scalar**. It will generally be of type `FloatTensor` or `LongTensor`. At the time of writing, PyTorch does not have a special tensor with zero dimensions. So, we use a one-dimension tensor with one element, as follows:

```
x = torch.rand(10)
x.size()

Output - torch.Size([10])
```

Vectors (1-D tensors)

A `vector` is simply an array of elements. For example, we can use a vector to store the average temperature for the last week:

```
temp = torch.FloatTensor([23,24,24.5,26,27.2,23.0])
temp.size()

Output - torch.Size([6])
```

Matrix (2-D tensors)

Most of the structured data is represented in the form of tables or matrices. We will use a dataset called `Boston House Prices`, which is readily available in the Python scikit-learn machine learning library. The dataset is a `numpy` array consisting of `506` samples or rows and `13` features representing each sample. Torch provides a utility function called `from_numpy()`, which converts a `numpy` array into a `torch` tensor. The shape of the resulting tensor is `506` rows x `13` columns:

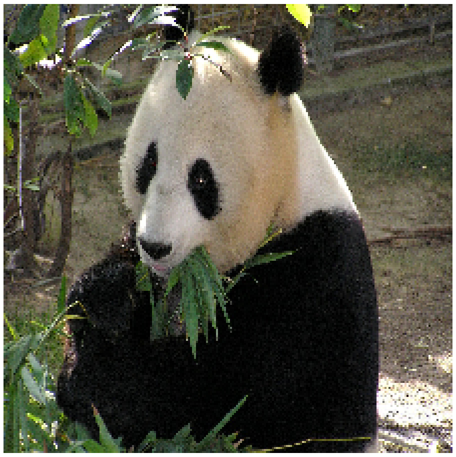
```
boston_tensor = torch.from_numpy(boston.data)
boston_tensor.size()

Output: torch.Size([506, 13])

boston_tensor[:2]

Output:
Columns 0 to 7
   0.0063 18.0000 2.3100 0.0000 0.5380 6.5750 65.2000 4.0900
   0.0273 0.0000 7.0700 0.0000 0.4690 6.4210 78.9000 4.9671

Columns 8 to 12
   1.0000 296.0000 15.3000 396.9000 4.9800
   2.0000 242.0000 17.8000 396.9000 9.1400
[torch.DoubleTensor of size 2x13]
```

3-D tensors

When we add multiple matrices together, we get a *3-D tensor*. 3-D tensors are used to represent data-like images. Images can be represented as numbers in a matrix, which are stacked together. An example of an image shape is `224`, `224`, `3`, where the first index represents height, the second represents width, and the third represents a channel (RGB). Let's see how a computer sees a panda, using the next code snippet:

```
from PIL import Image
# Read a panda image from disk using a library called PIL and convert it to r
panda = np.array(Image.open('panda.jpg').resize((224,224)))
panda_tensor = torch.from_numpy(panda)
panda_tensor.size()

Output - torch.Size([224, 224, 3])
#Display panda
plt.imshow(panda)
```

Since displaying the tensor of size `224`, `224`, `3` would occupy a couple of pages in the book, we will display the image and learn to slice the image into smaller tensors to visualize it:



Displaying the image

Slicing tensors

A common thing to do with a tensor is to slice a portion of it. A simple example could be choosing the first five elements of a one-dimensional tensor; let's call the tensor `sales`. We use a simple notation, `sales[:slice_index]` where `slice_index` represents the index where you want to slice the tensor:

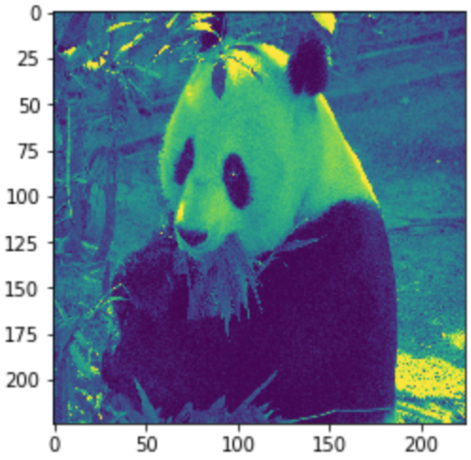
```
sales = torch.FloatTensor([1000.0,323.2,333.4,444.5,1000.0,323.2,333.4,444.5]

sales[:5]
 1000.0000
  323.2000
  333.4000
  444.5000
 1000.0000
[torch.FloatTensor of size 5]

sales[:-5]
 1000.0000
  323.2000
  333.4000
[torch.FloatTensor of size 3]
```

Let's do more interesting things with our panda image, such as see what the panda image looks like when only one channel is chosen and see how to select the face of the panda.

Here, we select only one channel from the panda image:

```
plt.imshow(panda_tensor[:,:,0].numpy())
#0 represents the first channel of RGB
```

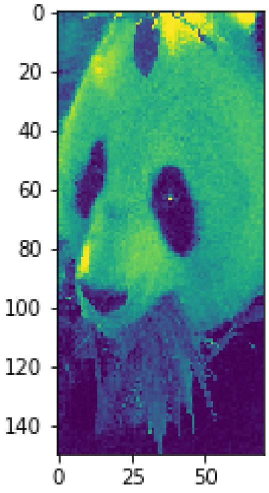
The output is as follows:


0
25
50
75
100
125
150
175
200
0
50
100
150
200

Now, lets crop the image. Say we want to build a face detector for pandas and we need just the face of a panda for that. We crop the tensor image such that it contains only the panda's face:

```
plt.imshow(panda_tensor[25:175,60:130,0].numpy())
```

The output is as follows:


0
20
40
60
80
100
120
140
0
25
50

Another common example would be where you need to pick a specific element of a tensor:

```
#torch.eye(shape) produces an diagonal matrix with 1 as it diagonal #elements
sales = torch.eye(3,3)
sales[0,1]

Output- 0.00.0
```

We will revisit image data in `Chapter 5`, *Deep Learning for Computer Vision,* when we discuss using CNNs to build image classifiers.

> *Most of the PyTorch tensor operations are very similar to* `NumPy` *operations.*

4-D tensors

One common example for four-dimensional tensor types is a batch of images. Modern CPUs and GPUs are optimized to perform the same operations on multiple examples faster. So, they take a similar time to process one image or a batch of images. So, it is common to use a batch of examples rather than use a single image at a time. Choosing the batch size is not straightforward; it depends on several factors. One major restriction for using a bigger batch or the complete dataset is GPU memory limitations—*16*, *32*, and *64* are commonly used batch sizes.

Let's look at an example where we load a batch of cat images of size `64` x `224` x `224` x `3` where *64* represents the batch size or the number of images, *244* represents height and width, and *3* represents channels:

```
#Read cat images from disk
cats = glob(data_path+'*.jpg')
#Convert images into numpy arrays
cat_imgs = np.array([np.array(Image.open(cat).resize((224,224))) for cat in c
cat_imgs = cat_imgs.reshape(-1,224,224,3)
cat_tensors = torch.from_numpy(cat_imgs)
cat_tensors.size()

Output - torch.Size([64, 224, 224, 3])
```

5-D tensors

One common example where you may have to use a five-dimensional tensor is video data. Videos can be split into frames, for example, a 30-second video containing a panda playing with a ball may contain 30 frames, which could be represented as a tensor of shape (1 x 30 x 224 x 224 x 3). A batch of such videos can be represented as tensors of shape (32 x 30 x 224 x 224 x 3)—*30* in the example represents, number of frames in that single video clip, where *32* represents the number of such video clips.

Tensors on GPU

We have learned how to represent different forms of data in tensor representation. Some of the common operations we perform once we have data in the form of tensors are addition, subtraction, multiplication, dot product, and matrix multiplication. All of these operations can be either performed on the CPU or the GPU. PyTorch provides a simple function called `cuda()` to copy a tensor on the CPU to the GPU. We will take a look at some of the operations and compare the performance between matrix multiplication operations on the CPU and GPU.

Tensor addition can be obtained by using the following code:

```
#Various ways you can perform tensor addition
a = torch.rand(2,2)
b = torch.rand(2,2)
c = a + b
d = torch.add(a,b)
#For in-place addition
a.add_(5)

#Multiplication of different tensors

a*b
a.mul(b)
#For in-place multiplication
a.mul_(b)
```

For tensor matrix multiplication, lets compare the code performance on CPU and GPU. Any tensor can be moved to the GPU by calling the `.cuda()` function.

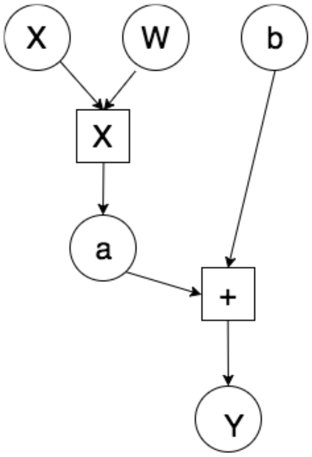
Multiplication on the GPU runs as follows:

```
a = torch.rand(10000,10000)
b = torch.rand(10000,10000)

a.matmul(b)

Time taken: 3.23 s

#Move the tensors to GPU
a = a.cuda()
b = b.cuda()

a.matmul(b)

Time taken: 11.2 µs
```

These fundamental operations of addition, subtraction, and matrix multiplication can be used to build complex operations, such as a **Convolution Neural Network** (**CNN**) and a **recurrent neural network** (**RNN**), which we will learn about in the later chapters of the book.

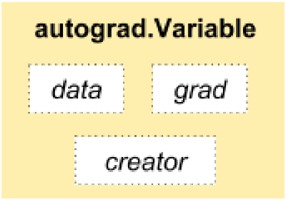
Variables

Deep learning algorithms are often represented as computation graphs. Here is a simple example of the variable computation graph that we built in our example:



Variable computation graph

Each circle in the preceding computation graph represents a variable. A variable forms a thin wrapper around a tensor object, its gradients, and a reference to the function that created it. The following figure shows `Variable` class components:



Variable class

The gradients refer to the rate of the change of the `loss` function with respect to various parameters (**W**, **b**). For example, if the gradient of **a** is 2, then any change in the value of **a** would modify the value of

Y by two times. If that is not clear, do not worry—most of the deep learning frameworks take care of calculating gradients for us. In this chapter, we learn how to use these gradients to improve the performance of our model.

Apart from gradients, a variable also has a reference to the function that created it, which in turn refers to how each variable was created. For example, the variable `a` has information that it is generated as a result of the product between `x` and `w`.

Let's look at an example where we create variables and check the gradients and the function reference:

```
x = Variable(torch.ones(2,2),requires_grad=True)
y = x.mean()

y.backward()

x.grad
Variable containing:
 0.2500  0.2500
 0.2500  0.2500
[torch.FloatTensor of size 2x2]

x.grad_fn
Output - None

x.data
 1 1
 1 1
[torch.FloatTensor of size 2x2]

y.grad_fn
<torch.autograd.function.MeanBackward at 0x7f6ee5cfc4f8>
```

In the preceding example, we called a `backward` operation on the variable to compute the gradients. By default, the gradients of the variables are none.

The `grad_fn` of the variable points to the function it created. If the variable is created by a user, like the variable `x` in our case, then the function reference is `None`. In the case of variable `y,` it refers to its function reference, `MeanBackward`.

The Data attribute accesses the tensor associated with the variable.

Creating data for our neural network

The `get_data` function in our first neural network code creates two variables, `x` and `y`, of sizes (`17`, `1`) and (`17`). We will take a look at what happens inside the function:

```
def get_data():
    train_X = np.asarray([3.3,4.4,5.5,6.71,6.93,4.168,9.779,6.182,7.59,2.167,
                         7.042,10.791,5.313,7.997,5.654,9.27,3.1])
    train_Y = np.asarray([1.7,2.76,2.09,3.19,1.694,1.573,3.366,2.596,2.53,1.2
                         2.827,3.465,1.65,2.904,2.42,2.94,1.3])
    dtype = torch.FloatTensor
    X = Variable(torch.from_numpy(train_X).type(dtype),requires_grad=False).v
    y = Variable(torch.from_numpy(train_Y).type(dtype),requires_grad=False)
    return X,y
```

Creating learnable parameters

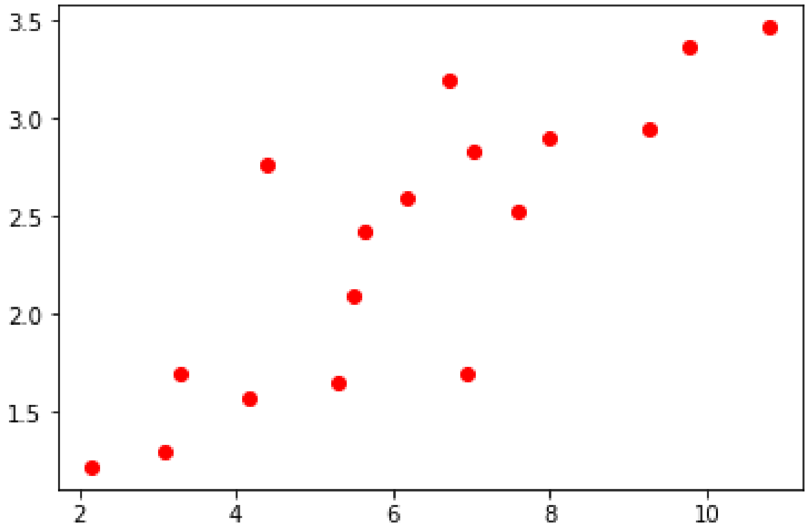
In our neural network example, we have two learnable parameters, `w` and `b`, and two fixed parameters, `x` and `y`. We have created variables `x` and `y` in our `get_data` function. Learnable parameters are created using random initialization and have the `require_grad` parameter set to `True`, unlike `x` and `y`, where it is set to `False`. There are different practices for initializing learnable parameters, which we will explore in the coming chapters. Let's take a look at our `get_weights` function:

```
def get_weights():
    w = Variable(torch.randn(1),requires_grad = True)
    b = Variable(torch.randn(1),requires_grad=True)
    return w,b
```

Most of the preceding code is self-explanatory; `torch.randn` creates a random value of any given shape.

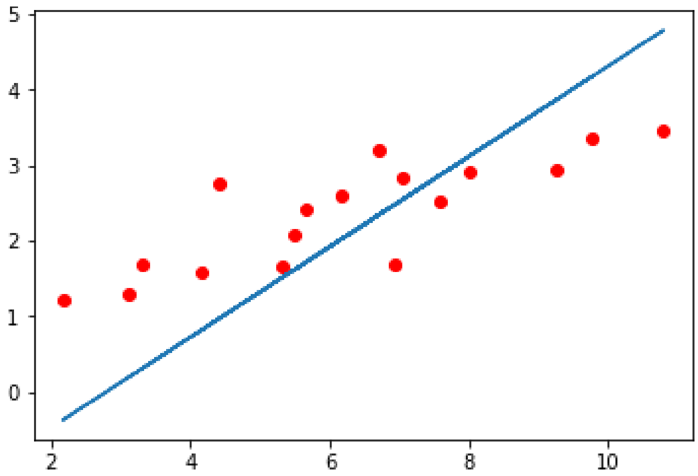
Neural network model

Once we have defined the inputs and outputs of the model using PyTorch variables, we have to build a model which learns how to map the outputs from the inputs. In traditional programming, we build a function by hand coding different logic to map the inputs to the outputs. However, in deep learning and machine learning, we learn the function by showing it the inputs and the associated outputs. In our example, we implement a simple neural network which tries to map the inputs to outputs, assuming a linear relationship. The linear relationship can be represented as $y = wx + b$, where w and b are learnable parameters. Our network has to learn the values of w and b, so that $wx + b$ will be closer to the actual y. Let's visualize our training dataset and the model that our neural network has to learn:



Input data points

The following figure represents a linear model fitted on input data points:



Linear model fitted on input data points

The dark-gray (blue) line in the image represents the model that our network learns.

Network implementation

As we have all the parameters (`x`, `w`, `b`, and `y`) required to implement the network, we perform a matrix multiplication between `w` and `x`. Then, sum the result with `b`. That will give our predicted `y`. The function is implemented as follows:

```
def simple_network(x):
    y_pred = torch.matmul(x,w)+b
    return y_pred
```

PyTorch also provides a higher-level abstraction in `torch.nn` called **layers**, which will take care of most of these underlying initialization and operations associated with most of the common techniques available in the neural network. We are using the lower-level operations to understand what happens inside these functions. In later chapters, that is `Chapter 5`, Deep Learning for Computer Vision and `Chapter 6`, Deep Learning with Sequence Data and Text, we will be relying on the PyTorch abstractions to build complex neural networks or functions. The previous model can be represented as a `torch.nn` layer, as follows:

```
f = nn.Linear(17,1) # Much simpler.
```

Now that we have calculated the `y` values, we need to know how good our model is, which is done in the `loss` function.

Loss function

As we start with random values, our learnable parameters, `w` and `b`, will result in `y_pred`, which will not be anywhere close to the actual `y`. So, we need to define a function which tells the model how close its predictions are to the actual values. Since this is a regression problem, we use a loss function called **sum of squared error** (**SSE**). We take the difference between the predicted `y` and the actual `y` and square it. SSE helps the model to understand how close the predicted values are to the actual values. The `torch.nn` library has different loss functions, such as MSELoss and cross-entropy loss. However, for this chapter, let's implement the `loss` function ourselves:

```
def loss_fn(y,y_pred):
    loss = (y_pred-y).pow(2).sum()
    for param in [w,b]:
        if not param.grad is None: param.grad.data.zero_()
    loss.backward()
    return loss.data[0]
```

Apart from calculating the loss, we also call the `backward` operation, which calculates the gradients of our learnable parameters, `w` and `b`. As we will use the `loss` function more than once, we remove any previously calculated gradients by calling the `grad.data.zero_()` operation. The first time we call the `backward` function, the gradients are empty, so we zero the gradients only when they are not `None`.

Optimize the neural network

We started with random weights to predict our targets and calculate loss for our algorithm. We calculate the gradients by calling the `backward` function on the final `loss` variable. This entire process repeats for one epoch, that is, for the entire set of examples. In most of the real-world examples, we will do the optimization step per iteration, which is a small subset of the total set. Once the loss is calculated, we optimize the values with the calculated gradients so that the loss reduces, which is implemented in the following function:

```
def optimize(learning_rate):
    w.data -= learning_rate * w.grad.data
    b.data -= learning_rate * b.grad.data
```

The learning rate is a hyper-parameter, which allows us to adjust the values in the variables by a small amount of the gradients, where the gradients denote the direction in which each variable (`w` and `b`) needs to be adjusted.

Different optimizers, such as Adam, RmsProp, and SGD are already implemented for use in the `torch.optim` package. We will be making use of these optimizers in later chapters to reduce the loss or improve the accuracy.

Loading data

Preparing data for deep learning algorithms could be a complex pipeline by itself. PyTorch provides many utility classes that abstract a lot of complexity such as data-parallelization through multi-threading, data-augmenting, and batching. In this chapter, we will take a look at two of the important utility classes, namely the `Dataset` class and the `DataLoader` class. To understand how to use these classes, let's take the `Dogs vs. Cats` dataset from Kaggle (`https://www.kaggle.com/c/dogs-vs-cats/data`) and create a data pipeline that generates a batch of images in the form of PyTorch tensors.

Dataset class

Any custom dataset class, say for example, our `Dogs` dataset class, has to inherit from the PyTorch dataset class. The custom class has to implement two main functions, namely `__len__(self)` and `__getitem__(self, idx)`. Any custom class acting as a `Dataset` class should look like the following code snippet:

```
from torch.utils.data import Dataset
class DogsAndCatsDataset(Dataset):
    def __init__(self,):
        pass
    def __len__(self):
        pass
    def __getitem__(self,idx):
        pass
```

We do any initialization, if required, inside the `init` method—for example, reading the index of the table and reading the filenames of the images, in our case. The `__len__(self)` operation is responsible for returning the maximum number of elements in our dataset. The `__getitem__(self, idx)` operation returns an element based on the `idx` every time it is called. The following code implements our `DogsAndCatsDataset` class:

```
class DogsAndCatsDataset(Dataset):

    def __init__(self,root_dir,size=(224,224)):
        self.files = glob(root_dir)
        self.size = size

    def __len__(self):
        return len(self.files)
    def __getitem__(self,idx):
        img = np.asarray(Image.open(self.files[idx]).resize(self.size))
        label = self.files[idx].split('/')[-2]
        return img,label
```

Once the `DogsAndCatsDataset` class is created, we can create an object and iterate over it, which is shown in the following code:

```
for image,label in dogsdset:
#Apply your DL on the dataset.
```

Applying a deep learning algorithm on a single instance of data is not optimal. We need a batch of data, as modern GPUs are optimized for better performance when executed on a batch of data.
The `DataLoader` class helps to create batches by abstracting a lot of complexity.

DataLoader class

The `DataLoader` class present in PyTorch's `utils` class combines a dataset object along with different samplers, such as `SequentialSampler` and `RandomSampler`, and provides us with a batch of images, either using a single or multi-process iterators. Samplers are different strategies for providing data to algorithms. The following is an example of a `DataLoader` for our `Dogs vs. Cats` dataset:

```
dataloader = DataLoader(dogsdset,batch_size=32,num_workers=2)
for imgs , labels in dataloader:
     #Apply your DL on the dataset.
     pass
```

`imgs` will contain a tensor of shape (32, 224, 224, 3), where *32* represents the batch size.

The PyTorch team also maintains two useful libraries, called `torchvision` and `torchtext`, which are built on top of the `Dataset` and `DataLoader` classes. We will use them in the relevant chapters.

Summary

In this chapter, we explored various data structures and operations provided by PyTorch. We implemented several components, using the fundamental blocks of PyTorch. For our data preparation, we created the tensors used by our algorithm. Our network architecture was a model for learning to predict average hours spent by users on our Wondermovies platform. We used the loss function to check the standard of our model and used the `optimize` function to adjust the learnable parameters of our model to make it perform better.

We also looked at how PyTorch makes it easier to create data pipelines by abstracting away several complexities that would require us to parallelize and augment data.

In the next chapter, we will dive deep into how neural networks and deep learning algorithms work. We will explore various PyTorch built-in modules for building network architectures, loss functions, and optimizations. We will also show how to use them on real-world datasets.

Diving Deep into Neural Networks

In this chapter, we will explore the different modules of deep learning architectures that are used to solve real-world problems. In the previous chapter, we used low-level operations of PyTorch to build modules such as a network architecture, a loss function, and an optimizer. In this chapter, we will explore some of the important components of neural networks required to solve real-world problems, along with how PyTorch abstracts away a lot of complexity by providing a lot of high-level functions. Towards the end of the chapter, we will build algorithms that solve real-world problems such as regression, binary classification, and multi-class classification.

In this chapter, we will go through following topics:

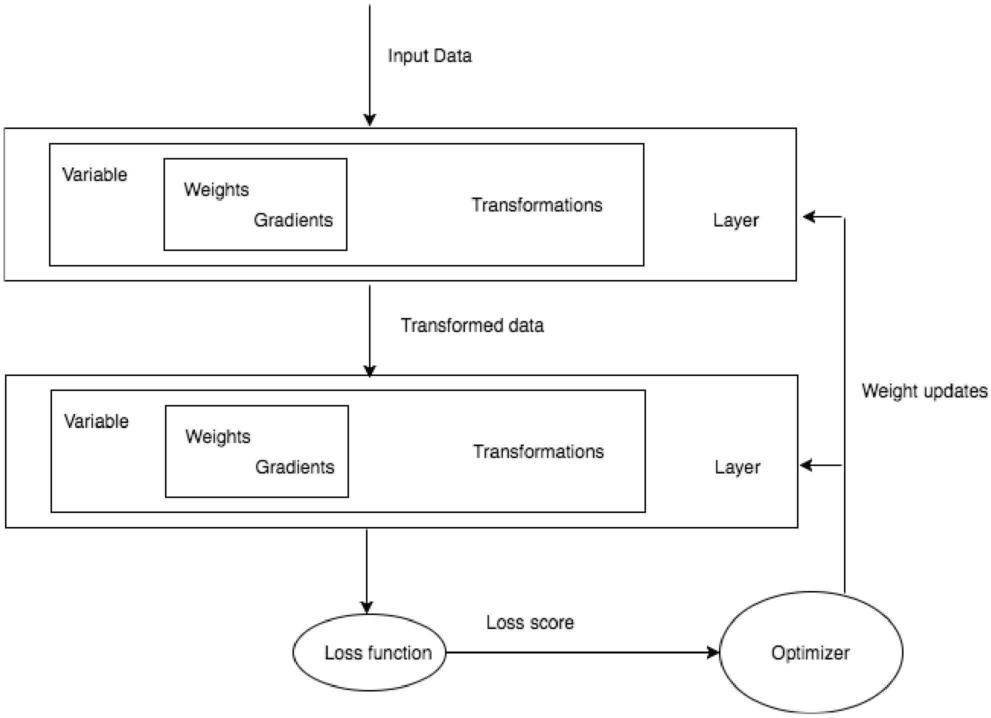
- Deep dive into the various building blocks of neural networks
- Exploring higher-level functionalities in PyTorch to build deep learning architectures
- Applying deep learning to a real-world image classification problem

Deep dive into the building blocks of neural networks

As we learned in the previous chapter, training a deep learning algorithm requires the following steps:

1. Building a data pipeline
2. Building a network architecture
3. Evaluating the architecture using a loss function
4. Optimizing the network architecture weights using an optimization algorithm

In the previous chapter, the network was composed of a simple linear model built using PyTorch numerical operations. Though building a neural architecture for a toy problem using numerical operations is easier, it quickly becomes complicated when we try to build architectures required to solve complex problems in different areas, such as computer vision and **natural language processing** (**NLP**). Most of the deep learning frameworks, such as PyTorch, TensorFlow, and Apache MXNet, provide higher-level functionalities that abstract a lot of this complexity. These higher-level functionalities are called **layers** across the deep learning frameworks. They accept input data, apply transformations like the ones we have seen in the previous chapter, and output the data. To solve real-world problems, deep learning architectures constitute of a number of layers ranging from 1 to 150, or sometimes more than that. Abstracting the low-level operations and training deep learning algorithms would look like the following diagram:


Input Data
Variable
Weights
Gradients
Transformations
Layer
Transformed data
Weight updates
Variable
Weights
Gradients
Transformations
Layer
Loss score
Loss function
Optimizer

Summarizing the previous diagram, any deep learning training involves getting data, building an architecture that in general is getting a bunch of layers together, evaluating the accuracy of the model using a loss function, and then optimizing the algorithm by optimizing the weights of our network. Before looking at solving some of the real-world problems, we will come to understand higher-level abstractions provided by PyTorch for building layers, loss functions, and optimizers.

Layers – fundamental blocks of neural networks

Throughout the rest of the chapter, we will come across different types of layers. To begin, let's try to understand one of the most important layers, the linear layer, which does exactly what our previous network architecture does. The linear layer applies a linear transformation:

$$Y = Wx + b$$

What makes it powerful is that fact that the entire function that we wrote in the previous chapter can be written in a single line of code, as follows:

```
from torch.nn import Linear
myLayer = Linear(in_features=10,out_features=5,bias=True)
```

The `myLayer` in the preceding code will accept a tensor of size `10` and outputs a tensor of size `5` after applying linear transformation. Let's look at a simple example of how to do that:

```
inp = Variable(torch.randn(1,10))
myLayer = Linear(in_features=10,out_features=5,bias=True)
myLayer(inp)
```

We can access the trainable parameters of the layer using the `weights` and `bias` attributes:

```
myLayer.weight

Output :
Parameter containing:
-0.2386 0.0828 0.2904 0.3133 0.2037 0.1858 -0.2642 0.2862 0.2874 0.1141
 0.0512 -0.2286 -0.1717 0.0554 0.1766 -0.0517 0.3112 0.0980 -0.2364 -0.0442
 0.0776 -0.2169 0.0183 -0.0384 0.0606 0.2890 -0.0068 0.2344 0.2711 -0.3039
 0.1055 0.0224 0.2044 0.0782 0.0790 0.2744 -0.1785 -0.1681 -0.0681 0.3141
 0.2715 0.2606 -0.0362 0.0113 0.1299 -0.1112 -0.1652 0.2276 0.3082 -0.2745
[torch.FloatTensor of size 5x10]

myLayer.bias

Output :
Parameter containing:
-0.2646
-0.2232
 0.2444
 0.2177
 0.0897
[torch.FloatTensor of size 5
```

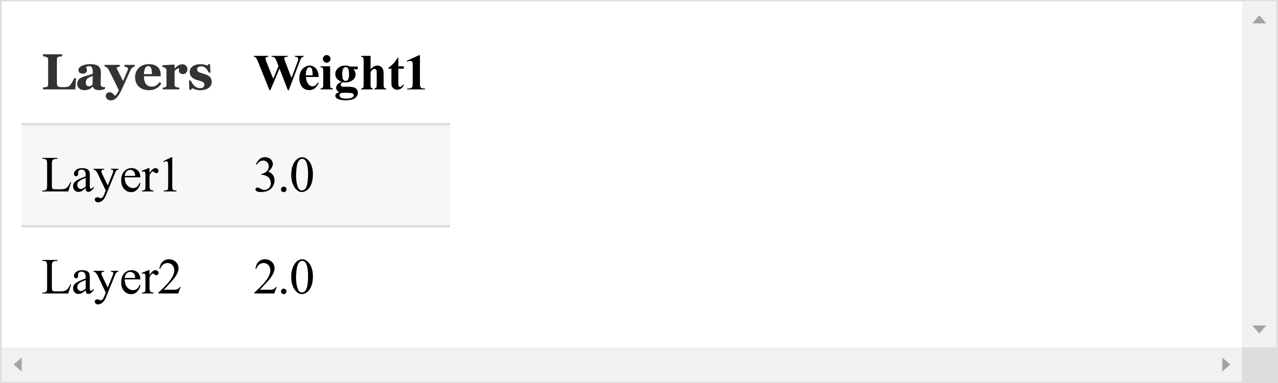
Linear layers are called by different names, such as **dense** or **fully connected layers** across different frameworks. Deep learning architectures used for solving real-world use cases generally contain more than one layer. In PyTorch, we can do it in multiple ways, shown as follows.

One simple approach is passing the output of one layer to another layer:

```
myLayer1 = Linear(10,5)
myLayer2 = Linear(5,2)
myLayer2(myLayer1(inp))
```

Each layer will have its own learnable parameters. The idea behind using multiple layers is that each layer will learn some kind of pattern that the later layers will build on. There is a problem in adding just linear layers together, as they fail to learn anything new beyond a simple representation of a linear layer. Let's see through a simple example of why it does not make sense to stack multiple linear layers together.

Let's say we have two linear layers with the following weights:



Layers	Weight1
Layer1	3.0
Layer2	2.0

The preceding architecture with two different layers can be simply represented as a single layer with a different layer. Hence, just stacking multiple linear layers will not help our algorithms to learn anything new. Sometimes, this can be unclear, so we can visualize the architecture with the following mathematical formulas:

$$Y = 2(3X_1) - 2\ Linear\ layers$$

$$Y = 6(X_1) - 1\ Linear\ layers$$

To solve this problem, we have different non-linearity functions that help in learning different relationships, rather than only focusing on linear relationships.

There are many different non-linear functions available in deep learning. PyTorch provides these non-linear functionalities as layers and we will be able to use them the same way we used the linear layer.

Some of the popular non-linear functions are as follows:

- Sigmoid
- Tanh
- ReLU
- Leaky ReLU

Non-linear activations

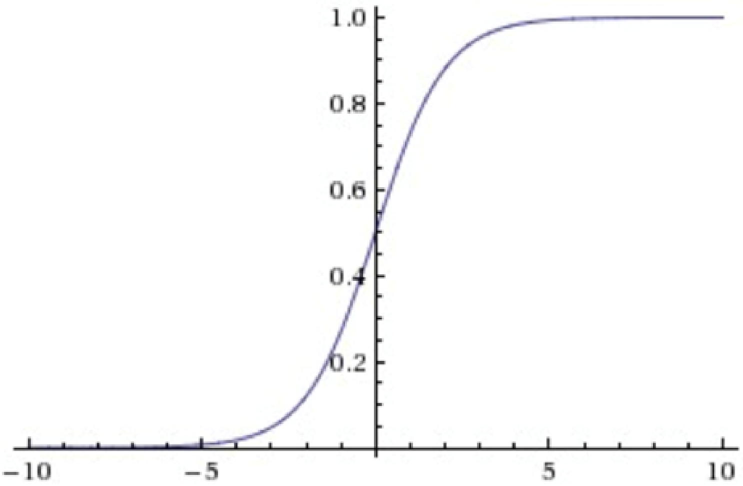
Non-linear activations are functions that take inputs and then apply a mathematical transformation and produce an output. There are several non-linear operations that we come across in practice. We will go through some of the popular non-linear activation functions.

Sigmoid

The sigmoid activation function has a simple mathematical form, as follows:

$$\sigma(x) = 1/(1 + e^{-x})$$

The sigmoid function intuitively takes a real-valued number and outputs a number in a range between zero and one. For a large negative number, it returns close to zero and, for a large positive number, it returns close to one. The following plot represents different sigmoid function outputs:



The sigmoid function has been historically used across different architectures, but in recent times it has gone out of popularity as it has one major drawback. When the output of the sigmoid function is close to zero or one, the gradients for the layers before the sigmoid function are close to zero and, hence, the learnable parameters of the previous layer get gradients close to zero and the weights do not get adjusted often, resulting in dead neurons.

Tanh

The tanh non-linearity function squashes a real-valued number in the range of -1 and 1. The tanh also faces the same issue of saturating gradients when tanh outputs extreme values close to -1 and 1. However, it is preferred to sigmoid, as the output of tanh is zero centered:

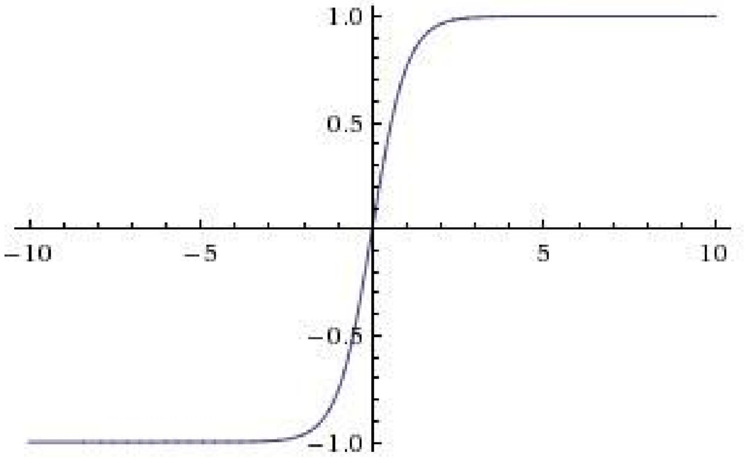


Image source: http://datareview.info/article/eto-nuzhno-znat-klyuchevyie-rekomendatsii-po-glubokomu-obucheniyu-chast-2/

ReLU

ReLU has become more popular in the recent years; we can find either its usage or one of its variants' usages in almost any modern architecture. It has a simple mathematical formulation:

$$f(x)=max(0,x)$$

In simple words, ReLU squashes any input that is negative to zero and leaves positive numbers as they are. We can visualize the ReLU function as follows:

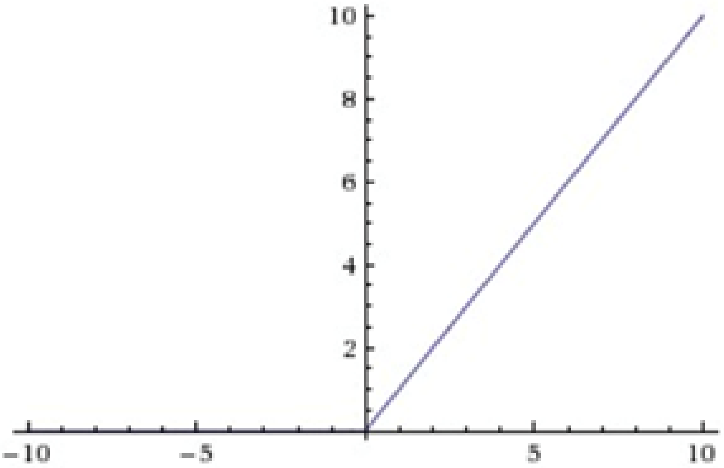


Image source: http://datareview.info/article/eto-nuzhno-znat-klyuchevyie-rekomendatsii-po-glubokomu-obucheniyu-chast-2/

Some of the pros and cons of using ReLU are as follows:

- It helps the optimizer in finding the right set of weights sooner. More technically it makes the convergence of stochastic gradient descent faster.

- It is computationally inexpensive, as we are just thresholding and not calculating anything like we did for the sigmoid and tangent functions.
- ReLU has one disadvantage; when a large gradient passes through it during the backward propagation, they often become non-responsive; these are called **dead neutrons**, which can be controlled by carefully choosing the learning rate. We will discuss how to choose learning rates when we discuss the different ways to adjust the learning rate in `Chapter 4`, *Fundamentals of Machine Learning*.

Leaky ReLU

Leaky ReLU is an attempt to solve a dying problem where, instead of saturating to zero, we saturate to a very small number such as 0.001. For some use cases, this activation function provides a superior performance to others, but it is not consistent.

PyTorch non-linear activations

PyTorch has most of the common non-linear activation functions implemented for us already and it can be used like any other layer. Let's see a quick example of how to use the `ReLU` function in PyTorch:

```
sample_data = Variable(torch.Tensor([[1,2,-1,-1]]))
myRelu = ReLU()
myRelu(sample_data)

Output:

Variable containing:
 1  2  0  0
[torch.FloatTensor of size 1x4]
```

In the preceding example, we take a tensor with two positive values and two negative values and apply a `ReLU` on it, which thresholds the negative numbers to `0` and retains the positive numbers as they are.

Now we have covered most of the details required for building a network architecture, let's build a deep learning architecture that can be used to solve real-world problems. In the previous chapter, we used a simple approach so that we could focus only on how a deep learning algorithm works. We will not be using that style to build our architecture anymore; rather, we will be building the architecture in the way it is supposed to be built in PyTorch.

The PyTorch way of building deep learning algorithms

All the networks in PyTorch are implemented as classes, subclassing a PyTorch class called `nn.Module`, and should implement `__init__` and `forward` methods. Inside the `init` function, we initialize any layers, such as the `linear` layer, which we covered in the previous section. In the `forward` method, we pass our input data into the layers that we initialized in our `init` method and return our final output. The non-linear functions are often directly used in the `forward` function and some use it in the `init` method too. The following code snippet shows how a deep learning architecture is implemented in PyTorch:

```
class MyFirstNetwork(nn.Module):

    def __init__(self,input_size,hidden_size,output_size):
        super(MyFirstNetwork,self).__init__()
        self.layer1 = nn.Linear(input_size,hidden_size)
        self.layer2 = nn.Linear(hidden_size,output_size)

    def __forward__(self,input):
        out = self.layer1(input)
        out = nn.ReLU(out)
        out = self.layer2(out)
        return out
```

If you are new to Python, some of the preceding code could be difficult to understand, but all it is doing is inheriting a parent class and implementing two methods in it. In Python, we subclass by passing the parent class as an argument to the class name. The `init` method acts as a constructor in Python and `super` is used to pass on arguments of the child class to the parent class, which in our case is `nn.Module`.

Model architecture for different machine learning problems

The kind of problem we are solving will decide mostly what layers we will use, starting from a linear layer to **Long Short-Term Memory** (**LSTM**) for sequential data. Based on the type of the problem you are trying to solve, your last layer is determined. There are three problems that we generally solve using any machine learning or deep learning algorithms. Let's look at what the last layer would look like:

- For a regression problem, such as predicting the price of a t-shirt to sell, we would use the last layer as a linear layer with an output of one, which outputs a continuous value.
- For classifying a given image as t-shirt or shirt, you would use a sigmoid activation function, as it outputs values either closer to one or zero, which is generally called a **binary classification problem**.
- For a multi-class classification, where we have to classify whether a given image is a t-shirt, jeans, shirt, or dress, we would use a softmax layer at the end our network. Let's try to understand intuitively what softmax does without going into the math of it. It takes inputs from the previous linear layer, for example, and outputs the probabilities for a given

number of examples. In our example, it would be trained to predict four probabilities for each type of image. Remember, all these probabilities always add up to one.

Loss functions

Once we have defined our network architecture, we are left with two important steps. One is calculating how good our network is at performing a particular task of regression, classification, and the next is optimizing the weight.

The optimizer (gradient descent) generally accepts a scalar value, so our `loss` function should generate a scalar value that has to be minimized during our training. Certain use cases, such as predicting where an obstacle is on the road and classifying it to a pedestrian or not, would require two or more loss functions. Even in such scenarios, we need to combine the losses to a single scalar for the optimizer to minimize. We will discuss examples of combining multiple losses to a single scalar in detail with a real-world example in the last chapter.

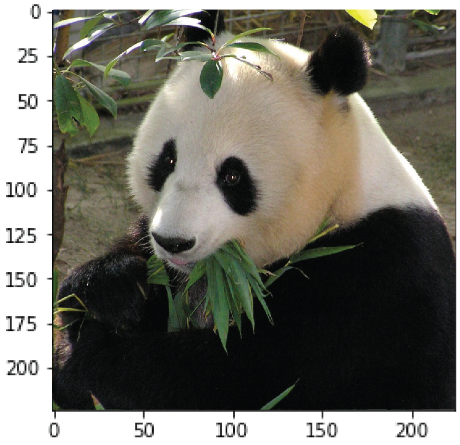
In the previous chapter, we defined our own `loss` function. PyTorch provides several implementations of commonly used `loss` functions. Let's take a look at the `loss` functions used for regression and classification.

The commonly used `loss` function for regression problems is **mean square error** (**MSE**). It is the same `loss` function we implemented in our previous chapter. We can use the `loss` function implemented in PyTorch, as follows:

```
loss = nn.MSELoss()
input = Variable(torch.randn(3, 5), requires_grad=True)
target = Variable(torch.randn(3, 5))
```

```
output = loss(input, target)
output.backward()
```

For classification, we use a cross-entropy loss. Before looking at the math for cross-entropy, let's understand what a cross-entropy loss does. It calculates the loss of a classification network predicting the probabilities, which should sum up to one, like our softmax layer. A cross-entropy loss increases when the predicted probability diverges from the correct probability. For example, if our classification algorithm predicts 0.1 probability for the following image to be a cat, but it is actually a panda, then the cross-entropy loss will be higher. If it predicts similar to the actual labels, then the cross-entropy loss will be lower:



Let's look at a sample implementation of how this actually happens in Python code:

```
def cross_entropy(true_label, prediction):
    if true_label == 1:
        return -log(prediction)
    else:
        return -log(1 - prediction)
```

To use a cross-entropy loss in a classification problem, we really do not need to be worried about what happens inside—all we have to remember is that, the loss will be high when our predictions are bad and low when predictions are good. PyTorch provides us with an implementation of the `loss`, which we can use, as follows:

```
loss = nn.CrossEntropyLoss()
input = Variable(torch.randn(3, 5), requires_grad=True)
target = Variable(torch.LongTensor(3).random_(5))
output = loss(input, target)
output.backward()
```

Some of the other `loss` functions that come as part of PyTorch are as follows:

L1 loss	Mostly used as a regularizer. We will discuss it further in `Chapter 4`, *Fundamentals of Machine Learning*.
MSE loss	Used as loss function for regression problems.
Cross-entropy loss	Used for binary and multi-class classification problems.
NLL Loss	Used for classification problems and allows us to use specific weights to handle imbalanced datasets.
NLL Loss2d	Used for pixel-wise classification, mostly for problems related to image segmentation.

Optimizing network architecture

Once we have calculated the loss of our network, we will optimize the weights to reduce the loss and thus improving the accuracy of the algorithm. For the sake of simplicity, let's see these optimizers as black boxes that take loss functions and all the learnable parameters and move them slightly to improve our performances. PyTorch provides most of the commonly used optimizers required in deep learning. If you want to explore what happens inside these optimizers and have a mathematical background, I would strongly recommend some of the following blogs:

- `http://colah.github.io/posts/2015-08-Backprop/`
- `http://ruder.io/deep-learning-optimization-2017/`

Some of the optimizers that PyTorch provides are as follows:

- ADADELTA
- Adagrad
- Adam
- SparseAdam
- Adamax
- ASGD

- LBFGS
- RMSProp
- Rprop
- SGD

We will get into the details of some of the algorithms in `Chapter 4`, *Fundamentals of Machine Learning*, along with some of the advantages and tradeoffs. Let's walk through some of the important steps in creating any `optimizer`:

```
optimizer = optim.SGD(model.parameters(), lr = 0.01)
```

In the preceding example, we created an `SGD` optimizer that takes all the learnable parameters of your network as the first argument and a learning rate that determines what ratio of change can be made to the learnable parameter. In `Chapter 4`, *Fundamentals of Machine Learning* we will get into more details of learning rates and momentum, which is an important parameter of optimizers. Once you create an optimizer object, we need to call `zero_grad()` inside our loop, as the parameters will accumulate the gradients created during the previous `optimizer` call:

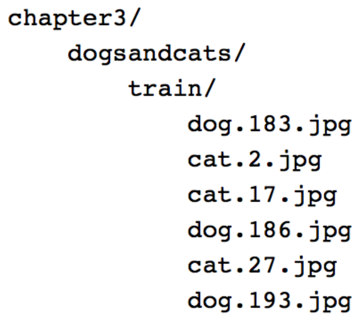
```
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

Once we call `backward` on the `loss` function, which calculates the gradients (quantity by which learnable parameters need to change), we call `optimizer.step()`, which makes the actual changes to our learnable parameter.

Now, we have covered most of the components required to help a computer see/ recognize images. Let's build a complex deep learning model that can differentiate between dogs and cats to put all the theory into practice.

Image classification using deep learning

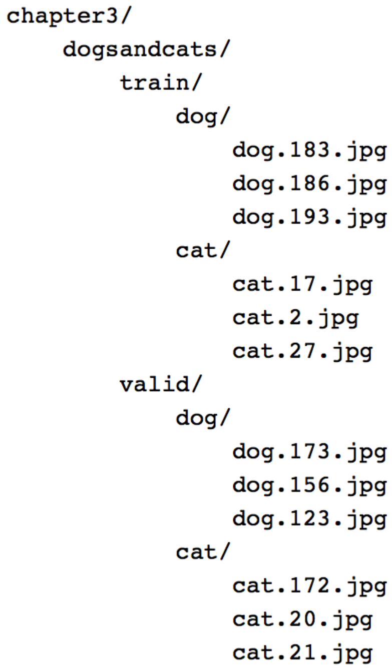
The most important step in solving any real-world problem is to get the data. Kaggle provides a huge number of competitions on different data science problems. We will pick one of the problems that arose in 2014, which we will use to test our deep learning algorithms in this chapter and improve it in `Chapter 5`, *Deep Learning for Computer Vision*, which will be on **Convolution Neural Networks** (**CNNs**) and some of the advanced techniques that we can use to improve the performance of our image recognition models. You can download the data from `https://www.kaggle.com/c/dogs-vs-cats/data`. The dataset contains 25,000 images of dogs and cats. Preprocessing of data and the creation of train, validation, and test splits are some of the important steps that need to be performed before we can implement an algorithm. Once the data is downloaded, taking a look at it, it shows that the folder contains images in the following format:



```
chapter3/
    dogsandcats/
        train/
            dog.183.jpg
            cat.2.jpg
            cat.17.jpg
            dog.186.jpg
            cat.27.jpg
            dog.193.jpg
```

Most of the frameworks make it easier to read the images and tag them to their labels when provided in the following format. That means that each class should have a separate folder of its images.

Here, all cat images should be in the `cat` folder and dog images in the `dog` folder:



```
chapter3/
    dogsandcats/
        train/
            dog/
                dog.183.jpg
                dog.186.jpg
                dog.193.jpg
            cat/
                cat.17.jpg
                cat.2.jpg
                cat.27.jpg
        valid/
            dog/
                dog.173.jpg
                dog.156.jpg
                dog.123.jpg
            cat/
                cat.172.jpg
                cat.20.jpg
                cat.21.jpg
```

Python makes it easy to put the data into the right format. Let's quickly take a look at the code and, then, we will go through the important parts of it:

```
path = '../chapter3/dogsandcats/'

#Read all the files inside our folder.
files = glob(os.path.join(path,'*/*.jpg'))

print(f'Total no of images {len(files)}')

no_of_images = len(files)

#Create a shuffled index which can be used to create a validation data set
shuffle = np.random.permutation(no_of_images)

#Create a validation directory for holding validation images.
os.mkdir(os.path.join(path,'valid'))

#Create directories with label names
for t in ['train','valid']:
     for folder in ['dog/','cat/']:
          os.mkdir(os.path.join(path,t,folder))

#Copy a small subset of images into the validation folder.
for i in shuffle[:2000]:
     folder = files[i].split('/')[-1].split('.')[0]
     image = files[i].split('/')[-1]
     os.rename(files[i],os.path.join(path,'valid',folder,image))

#Copy a small subset of images into the training folder.
for i in shuffle[2000:]:
     folder = files[i].split('/')[-1].split('.')[0]
     image = files[i].split('/')[-1]
     os.rename(files[i],os.path.join(path,'train',folder,image))
```

All the preceding code does is retrieve all the files and pick 2,000 images for creating a validation set. It segregates all the images into the two categories of cats and dogs. It is a common and important practice to create a separate validation set, as it is not fair to test our algorithms on the same data it is trained on. To create a `validation` dataset, we create a list of numbers that are in the range of the length of the images in a shuffled order. The shuffled numbers act as an index for us to pick a bunch of images for creating our `validation` dataset. Let's go through each section of the code in detail.

We create a file using the following code:

```
files = glob(os.path.join(path,'*/*.jpg'))
```

The `glob` method returns all the files in the particular path. When there are a huge number of images, we can also use `iglob`, which returns an iterator, instead of loading the names into memory. In our case, we have only 25,000 filenames, which can easily fit into memory.

We can shuffle our files using the following code:

```
shuffle = np.random.permutation(no_of_images)
```

The preceding code returns 25,000 numbers in the range from zero to 25,000 in a shuffled order, which we will use as an index for selecting a subset of images to create a `validation` dataset.

We can create a validation code, as follows:

```
os.mkdir(os.path.join(path,'valid'))
for t in ['train','valid']:
     for folder in ['dog/','cat/']:
          os.mkdir(os.path.join(path,t,folder))
```

The preceding code creates a `validation` folder and creates folders based on categories (cats and dogs) inside `train` and `valid` directories.

We can shuffle an index with the following code:

```
for i in shuffle[:2000]:
     folder = files[i].split('/')[-1].split('.')[0]
     image = files[i].split('/')[-1]
     os.rename(files[i],os.path.join(path,'valid',folder,image))
```

In the preceding code, we use our shuffled index to randomly pick `2000` different images for our validation set. We do something similar for the training data to segregate the images in the `train` directory.

As we have the data in the format we need, let's quickly look at how to load the images as PyTorch tensors.

Loading data into PyTorch tensors

The PyTorch `torchvision.datasets` package provides a utility class called `ImageFolder` that can be used to load images along with their associated labels when data is presented in the aforementioned format. It is a common practice to perform the following preprocessing steps:

1. Resize all the images to the same size. Most of the deep learning architectures expect the images to be of the same size.
2. Normalize the dataset with the mean and standard deviation of the dataset.
3. Convert the image dataset to a PyTorch tensor.

PyTorch makes a lot of these preprocessing steps easier by providing a lot of utility functions in the `transforms` module. For our example, let's apply three transformations:

- Scale to a 256 x 256 image size
- Convert to a PyTorch tensor
- Normalize the data (we will talk about how we arrived at the mean and standard deviation in `Chapter 5`, *Deep Learning for Computer Vision*)

The following code demonstrates how transformation can be applied and images are loaded using the `ImageFolder` class:

```
simple_transform=transforms.Compose([transforms.Scale((224,224)),
                             transforms.ToTensor(),
                             transforms.Normalize([0.485, 0.456,
train = ImageFolder('dogsandcats/train/',simple_transform)
valid = ImageFolder('dogsandcats/valid/',simple_transform)
```

The `train` object holds all the images and associated labels for the dataset. It contains two important attributes: one that gives a mapping between classes and the associated index used in the dataset and another one that gives a list of classes:

- `train.class_to_idx - {'cat': 0, 'dog': 1}`
- `train.classes - ['cat', 'dog']`

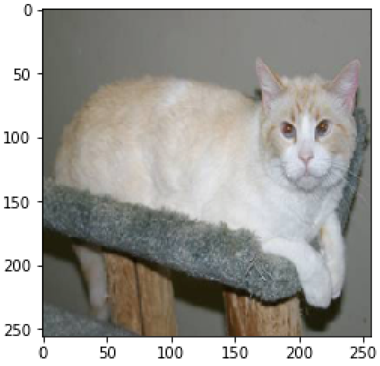
It is often a best practice to visualize the data loaded into tensors. To visualize the tensors, we have to reshape the tensors and denormalize the values. The following function does that for us:

```
def imshow(inp):
    """Imshow for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
```

Now, we can pass our tensor to the preceding `imshow` function, which converts it into an image:

```
imshow(train[50][0])
```

The preceding code generates the following output:


0
50
100
150
200
250
0
50
100
150
200
250

Loading PyTorch tensors as batches

It is a common practice in deep learning or machine learning to batch samples of images, as modern **graphics processing units** (**GPUs**) and CPUs are optimized to run operations faster on a batch of images. The batch size generally varies depending on the kind of GPU we use. Each GPU has its own memory, which can vary from 2 GB to 12 GB, and sometimes more for commercial GPUs. PyTorch provides the `DataLoader` class, which takes in a dataset and returns us a batch of images. It abstracts a lot of complexities in batching, such as the usage of multi-workers for applying transformation. The following code converts the previous `train` and `valid` datasets into data loaders:

```
train_data_gen =
  torch.utils.data.DataLoader(train,batch_size=64,num_workers=3)
valid_data_gen =
  torch.utils.data.DataLoader(valid,batch_size=64,num_workers=3)
```

The `DataLoader` class provides us with a lot of options and some of the most commonly used ones are as follows:

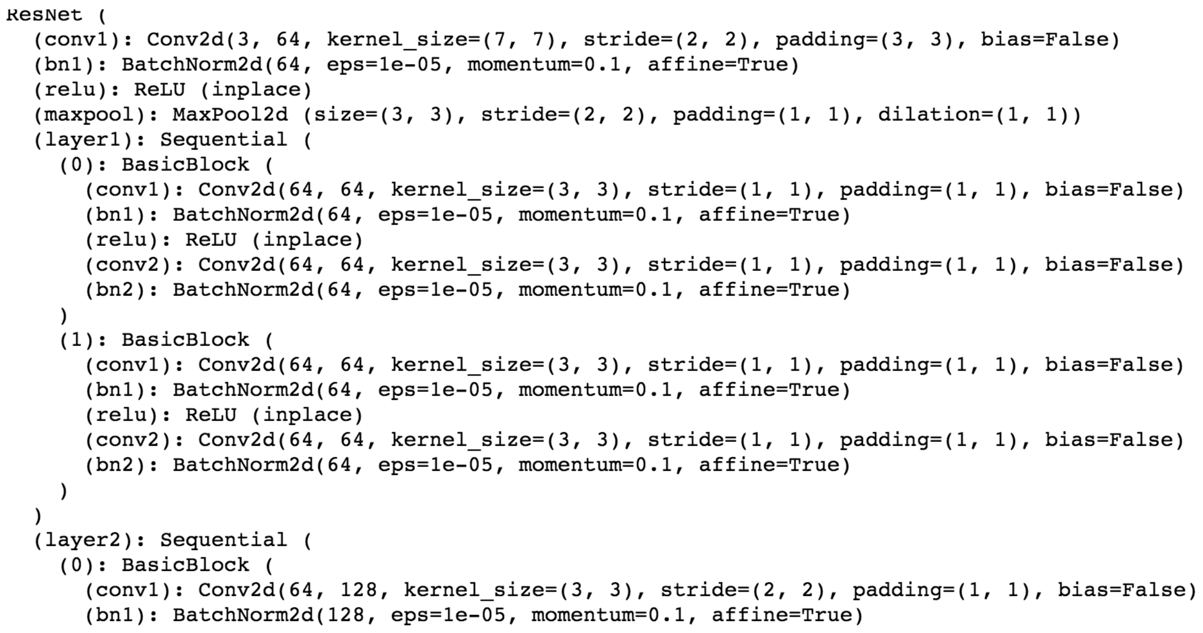
- `shuffle`: When true, this shuffles the images every time the data loader is called.
- `num_workers`: This is responsible for parallelization. It is common practice to use a number of workers fewer than the number of cores available in your machine.

Building the network architecture

For most of the real-world use cases, particularly in computer vision, we rarely build our own architecture. There are different architectures that can be quickly used to solve our real-world problems. For our example, we use a popular deep learning algorithm called **ResNet**, which won the first prize in 2015 in different competitions, such as ImageNet, related to computer vision. For a simpler understanding, let's assume that this algorithm is a bunch of different PyTorch layers carefully tied together and not focus on what happens inside this algorithm. We will see some of the key building blocks of the ResNet algorithm in `Chapter 5`, *Deep Learning for Computer Vision*, when we learn about CNNs. PyTorch makes it easier to use a lot of these popular algorithms by providing them off the shelf in the `torchvision.models` module. So, for this example, let's quickly take a look at how to use this algorithm and then walk through each line of code:

```
model_ft = models.resnet18(pretrained=True)
num_ftrs = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_ftrs, 2)

if is_cuda:
    model_ft = model_ft.cuda()
```

The `models.resnet18(pertrained = True)` object creates an instance of the algorithm, which is a collection of PyTorch layers. We can take a quick look at what constitutes the ResNet algorithm by printing `model_ft`. A small portion of the algorithm looks like the following screenshot. I am not including the full algorithm as it could run for several pages:



```
ResNet (
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
  (relu): ReLU (inplace)
  (maxpool): MaxPool2d (size=(3, 3), stride=(2, 2), padding=(1, 1), dilation=(1, 1))
  (layer1): Sequential (
    (0): BasicBlock (
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
      (relu): ReLU (inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
    )
    (1): BasicBlock (
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
      (relu): ReLU (inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
    )
  )
  (layer2): Sequential (
    (0): BasicBlock (
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True)
```

As we can see, the ResNet architecture is a collection of layers, namely `Conv2d`, `BatchNorm2d`, and `MaxPool2d`, stitched in a particular way. All these algorithms will accept an argument called **pretrained**. When `pretrained` is `True`, the weights of the algorithm are already tuned for a particular ImageNet classification problem of predicting 1,000 different categories, which include cars, ships, fish, cats, and dogs. This algorithm is trained to predict the 1,000 ImageNet categories and the weights are adjusted to a certain point where the algorithm achieves state-of-art accuracy. These weights are stored and shared with the model that we are using for the use case. Algorithms tend to work better when started with fine-tuned weights, rather than when started with random weights. So, for our use case, we start with pretrained weights.

The ResNet algorithm cannot be used directly, as it is trained to predict one of the 1,000 categories. For our use case, we need to predict only one of the two categories of dogs and cats. To achieve this, we take the last layer of the ResNet model, which is a `linear` layer and change the output features to two, as shown in the following code:

```
model_ft.fc = nn.Linear(num_ftrs, 2)
```

If you are running this algorithm on a GPU-based machine, then to make the algorithm run on a GPU we call the `cuda` method on the model. It is strongly recommended that you run these programs on a GPU-powered machine; it is easy to spin a cloud instance with a GPU for less than a dollar. The last line in the following code snippet tells PyTorch to run the code on the GPU:

```
if is_cuda:
    model_ft = model_ft.cuda()
```

Training the model

In the previous sections, we have created `DataLoader` instances and algorithms. Now, let's train the model. To do this we need a `loss` function and an `optimizer`:

```
# Loss and Optimizer
learning_rate = 0.001
criterion = nn.CrossEntropyLoss()
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7,
  gamma=0.1)
```

In the preceding code, we created our `loss` function based on `CrossEntropyLoss` and the optimizer based on `SGD`. The `StepLR` function helps in dynamically changing the learning rate. We will discuss different strategies available to tune the learning rate in `Chapter 4`, *Fundamentals of Machine Learning*.

The following `train_model` function takes in a model and tunes the weights of our algorithm by running multiple epochs and reducing the loss:

```
def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
    since = time.time()

    best_model_wts = model.state_dict()
    best_acc = 0.0

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)
        # Each epoch has a training and validation phase
        for phase in ['train', 'valid']:
            if phase == 'train':
                scheduler.step()
                model.train(True) # Set model to training mode
            else:
                model.train(False) # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for data in dataloaders[phase]:
                # get the inputs
                inputs, labels = data

                # wrap them in Variable
                if is_cuda:
                    inputs = Variable(inputs.cuda())
                    labels = Variable(labels.cuda())
                else:
                    inputs, labels = Variable(inputs),                 Variab

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                outputs = model(inputs)
                _, preds = torch.max(outputs.data, 1)
                loss = criterion(outputs, labels)

                # backward + optimize only if in training phase
                if phase == 'train':
                    loss.backward()
                    optimizer.step()

                # statistics
                running_loss += loss.data[0]
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects / dataset_sizes[phase]

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))

            # deep copy the model
            if phase == 'valid' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = model.state_dict()

        print()

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))

    # load best model weights
    model.load_state_dict(best_model_wts)
    return model
```

The preceding function does the following:

1. Passes the images through the model and calculates the loss.
2. Backpropagates during the training phase. For the validation/testing phase, it does not adjust the weights.
3. The loss is accumulated across batches for each epoch.
4. The best model is stored and validation accuracy is printed.

The preceding model, after running for `25` epochs, results in a validation accuracy of 87%. The following is the log generated by the preceding `train_model` function when run on our `Dogs vs. Cats` dataset; I am just including the result of the last few epochs to save space in the book:

```
Epoch 18/24
----------
train Loss: 0.0044 Acc: 0.9877
valid Loss: 0.0059 Acc: 0.8740

Epoch 19/24
----------
train Loss: 0.0043 Acc: 0.9914
```

```
valid Loss: 0.0059 Acc: 0.8725

Epoch 20/24
----------
train Loss: 0.0041 Acc: 0.9932
valid Loss: 0.0060 Acc: 0.8725

Epoch 21/24
----------
train Loss: 0.0041 Acc: 0.9937
valid Loss: 0.0060 Acc: 0.8725

Epoch 22/24
----------
train Loss: 0.0041 Acc: 0.9938
valid Loss: 0.0060 Acc: 0.8725

Epoch 23/24
----------
train Loss: 0.0041 Acc: 0.9938
valid Loss: 0.0060 Acc: 0.8725

Epoch 24/24
----------
train Loss: 0.0040 Acc: 0.9939
valid Loss: 0.0060 Acc: 0.8725

Training complete in 27m 8s
Best val Acc: 0.874000
```

In the coming chapters, we will learn more advanced techniques that will help us in training more accurate models in a much faster way. The preceding model took around 30 minutes to run on a Titan X GPU. We will cover different techniques that will help in training the model faster.

Summary

In this chapter, we explored the complete life cycle of a neural network in Pytorch, starting from constituting different types of layers, adding activations, calculating cross-entropy loss, and finally optimizing network performance (that is, minimizing loss), by adjusting the weights of layers using the SGD optimizer.

We have studied how to apply the popular ResNET architecture to binary or multi-class classification problems.

While doing this, we have tried to solve the real-world image classification problem of classifying a cat image as a cat and a dog image as a dog. This knowledge can be applied to classify different categories/classes of entities, such as classifying species of fish, identifying different kinds of dogs, categorizing plant seedlings, grouping together cervical cancer into Type 1, Type 2, and Type 3, and much more.

In the next chapter, we will go through the fundamentals of machine learning.

Fundamentals of Machine Learning

In the previous chapters, we saw practical examples of how to build deep learning models to solve classification and regression problems, such as image classification and average user view predictions. Similarly, we developed an intuition on how to frame a deep learning problem. In this chapter, we will take a look at how we can attack different kinds of problems and different tweaks that we will potentially end up using to improve our model's performance on our problems.

In this chapter, we will explore:

- Other forms of problems beyond classification and regression
- Problems with evaluation, understanding overfitting, underfitting, and techniques to solve them
- Preparing data for deep learning

Remember, most of the topics that we discuss in this chapter are common to machine learning and deep learning, except for some of the techniques—such as dropout—that we use to solve overfitting problems.

Three kinds of machine learning problems

In all our previous examples, we tried to solve either classification (predicting cats or dogs) or regression (predicting the average time users spend in the platform) problems. All these are examples of supervised learning, where the goal is to map the relationship between training examples and their targets and use it to make predictions on unseen data.

Supervised learning is just one part of machine learning, and there are other different parts of machine learning. There are three different kinds of machine learning:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

Let's look in detail at the kinds of algorithms.

Supervised learning

Most of the successful use cases in the deep learning and machine learning space fall under supervised learning. Most of the examples we cover in this book will also be part of this. Some of the common examples of supervised learning are:

- **Classification problems**: Classifying dogs and cats.
- **Regression problems**: Predicting stock prices, cricket match scores, and so on.
- **Image segmentation**: Doing a pixel-level classification. For a self-driving car, it is important to identify what each pixel belongs to from the photo taken by its camera. The pixel could belong to a car, pedestrian, tree, bus, and so on.
- **Speech recognition**: OK Google, Alexa, and Siri are good examples of speech recognition.
- **Language translation**: Translating speech from one language to another language.

Unsupervised learning

When there is no label data, unsupervised learning techniques help in understanding the data by visualizing and compressing. The two commonly-used techniques in unsupervised learning are:

- Clustering
- Dimensionality reduction

Clustering helps in grouping all similar data points together. Dimensionality reduction helps in reducing the number of dimensions, so that we can visualize high-dimensional data to find any hidden patterns.

Reinforcement learning

Reinforcement learning is the least popular machine learning category. It did not find its success in real-world use cases. However, it has changed in recent years, and teams from Google DeepMind were able to successfully build systems based on reinforcement learning and were able to win the AlphaGo game against the world champion. This kind of technology advancement, where a computer can beat a human in a game, was considered to take more than a few decades for computers to achieve. However, deep learning combined with reinforcement learning was able to achieve it far sooner than anyone would have anticipated. These techniques have started seeing early success, and it could probably take a few years for it to become mainstream.

In this book, we will focus mostly on the supervised techniques and some of the unsupervised techniques that are specific to deep learning, such as generative networks used for creating images of a particular style called **style transfer** and **generative adversarial networks**.

Machine learning glossary

In the last few chapters, we have used lot of terminology that could be completely new to you if you are just entering the machine learning or deep learning space. We will list a lot of commonly-used terms in machine learning, which are also used in the deep learning literature:

- **Sample or input or data point**: These mean particular instances of training a set. In our image classification problem seen in the last chapter, each image can be referred to as a sample, input, or data point.
- **Prediction or output**: The value our algorithm generates as an output. For example, in our previous example our algorithm predicted a particular image as 0, which is the label given to cat, so the number 0 is our prediction or output.
- **Target or label**: The actual tagged label for an image.
- **Loss value or prediction error**: Some measure of distance between the predicted value and actual value. The smaller the value, the better the accuracy.
- **Classes**: Possible set of values or labels for a given dataset. In the example in our previous chapter, we had two classes —cats and dogs.

- **Binary classification**: A classification task where each input example should be classified as either one of the two exclusive categories.
- **Multi-class classification**: A classification task where each input example can be classified into of more than two different categories.
- **Multi-label classification**: An input example can be tagged with multiple labels—for example, tagging a restaurant with different types of food it serves such as Italian, Mexican, and Indian. Another commonly-used example is object detection in an image, where the algorithm identifies different objects in the image.
- **Scalar regression**: Each input data point will be associated with one scalar quality, which is a number. Some examples could be predicting house prices, stock prices, and cricket scores.
- **Vector regression**: Where the algorithm needs to predict more than one scalar quantity. One good example is when you try to identify the bounding box that contains the location of a fish in an image. In order to predict the bounding box, your algorithm needs to predict four scalar quantities denoting the edges of a square.
- **Batch**: For most cases, we train our algorithm on a bunch of input samples referred to as the batch. The batch size varies generally from 2 to 256, depending on the GPU's memory. The weights are also updated for each batch, so the

algorithms tend to learn faster than when trained on a single example.

- **Epoch**: Running the algorithm through a complete dataset is called an **epoch**. It is common to train (update the weights) for several epochs.

Evaluating machine learning models

In the example of image classification that we covered in the last chapter, we split the data into two different halves, one for training and one for validation. It is a good practice to use a separate dataset to test the performance of your algorithm, as testing the algorithm on the training set may not give you the true generalization power of the algorithm. In most real-world use cases, based on the validation accuracy, we often tweak our algorithm in different ways, such as adding more layers or different layers, or using different techniques that we will cover in the later part of the chapter. So, there is a higher chance that your choices for tweaking the algorithm are based on the validation dataset. Algorithms trained this way tend to perform well in the training dataset and the validation dataset, but fail to generalize well on unseen data. This is due to an information leak from your validation dataset, which influences us in tweaking the algorithm.

To avoid the problem of an information leak and improve generalization, it is often a common practice to split the dataset into three different parts, namely a training, validation, and test dataset. We do the training and do all the hyper parameter tuning of the algorithm using the training and validation set. At the end, when the entire training is done, then you will test the algorithm on the test dataset. There are two types of parameters that we talk about. One is the parameters or weights that are used inside an algorithm, which are tuned by the optimizer or during backpropagation. The other set of parameters, called **hyper parameters**, controls the number of layers used in the network, learning rate, and other types of parameter that generally change the architecture, which is often done manually.

The phenomenon of a particular algorithm performing better in the training set and failing to perform on the validation or test set is called **overfitting**, or the lack of the algorithm's ability to generalize. There is an opposite phenomenon where the algorithm fails to perform for the training set, which is called **underfitting**. We will look at different strategies that will help us in overcoming the overfitting and underfitting problems.

Let's look at the various strategies available for splitting the dataset before looking at overfitting and underfitting.

Training, validation, and test split

It is best practice to split the data into three parts—training, validation, and test datasets. The best approach for using the holdout dataset is to:

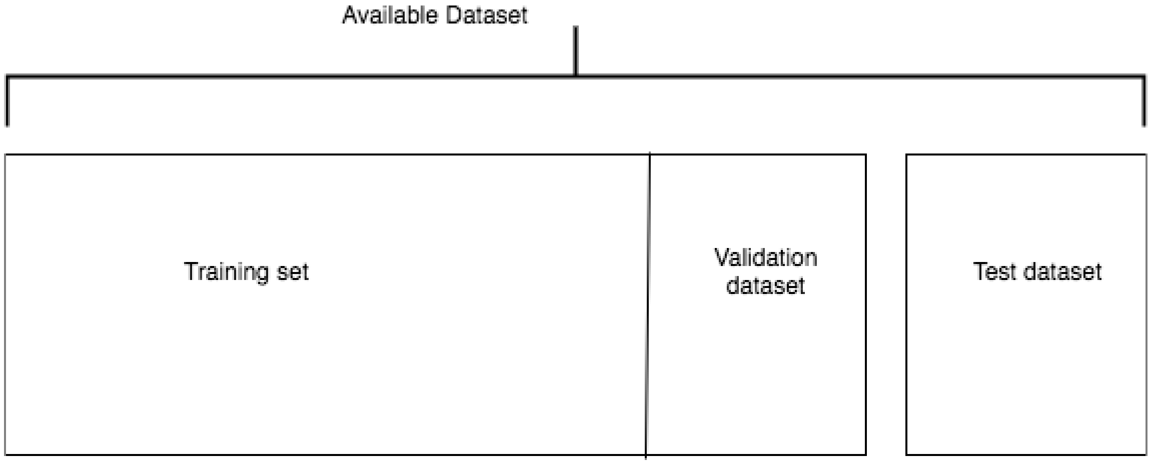
1. Train the algorithm on the training dataset
2. Perform hyper parameter tuning based on the validation dataset
3. Perform the first two steps iteratively until the expected performance is achieved
4. After freezing the algorithm and the hyper parameters, evaluate it on the test dataset

Avoid splitting the data into two parts, as it may lead to an information leak. Training and testing it on the same dataset is a clear no-no as it does not guarantee algorithm generalization. There are three popular holdout strategies that can be used to split the data into training and validation sets. They are as follows:

- Simple holdout validation
- K-fold validation
- Iterated k-fold validation

Simple holdout validation

Set apart a fraction of the data as your test dataset. What fraction to keep may be very problem-specific and could largely depend on the amount of data available. For problems particularly in the fields of computer vision and NLP, collecting labeled data could be very expensive, so to hold out a large fraction of 30% may make it difficult for the algorithm to learn, as it will have less data to train on. So, depending on the data availability, choose the fraction of it wisely. Once the test data is split, keep it apart until you freeze the algorithm and its hyper parameters. For choosing the best hyper parameters for the problem, choose a separate validation dataset. To avoid overfitting, we generally divide available data into three different sets, as shown in following image:



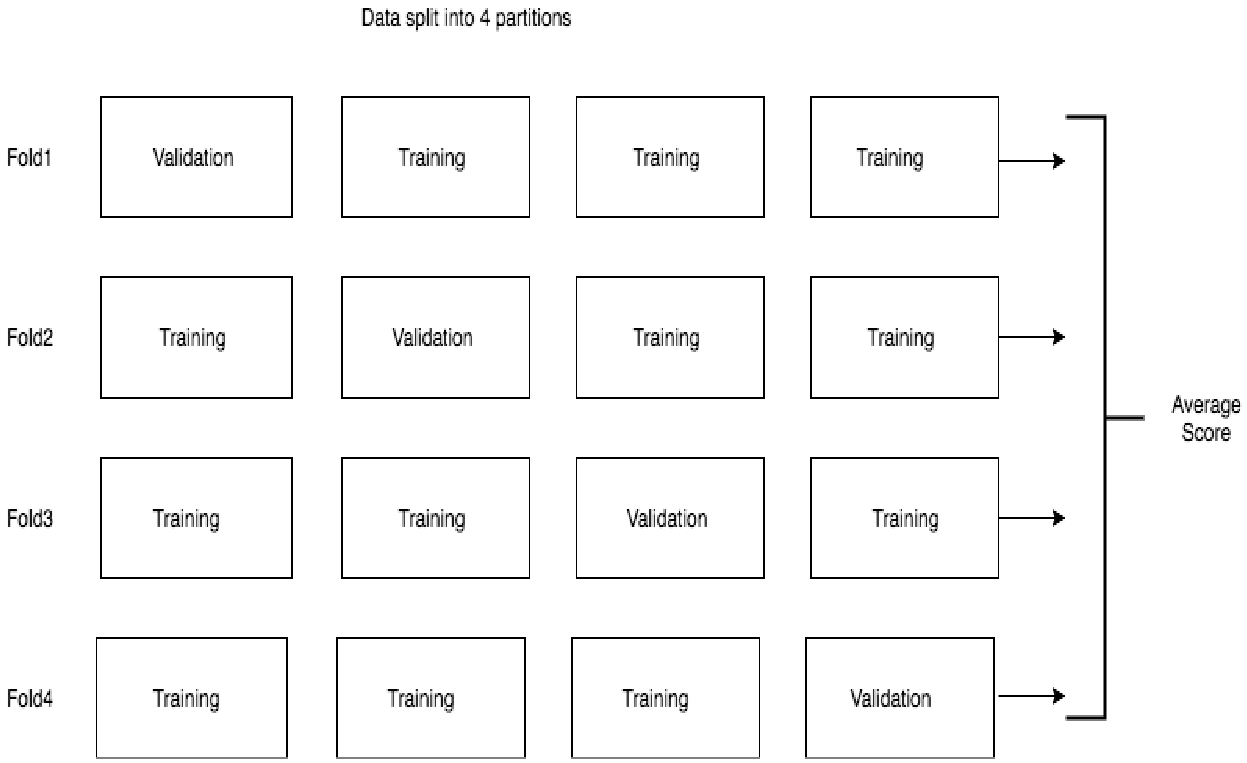
We used a simple implementation of the preceding figure in the last chapter to create our validation set. Let's look at a snapshot of the implementation:

```
files = glob(os.path.join(path,'*/*.jpg'))
no_of_images = len(files)
shuffle = np.random.permutation(no_of_images)
train = files[shuffle[:int(no_of_images*0.8)]]
valid = files[shuffle[int(no_of_images*0.8):]]
```

This is one of the simplest holdout strategies and is commonly used to start with. There is a disadvantage of using this with small datasets. The validation dataset or test dataset may not be statistically representative of the data at hand. We can easily recognize this by shuffling the data before holding out. If the results obtained are not consistent, then we need to use a better approach. To avoid this issue, we often end up using k-fold or iterated k-fold validation.

K-fold validation

Keep a fraction of the dataset for the test split, then divide the entire dataset into k-folds where k can be any number, generally varying from two to ten. At any given iteration, we hold one block for validation and train the algorithm on the rest of the blocks. The final score is generally the average of all the scores obtained across the k-folds. The following diagram shows an implementation of k-fold validation where k is four; that is, the data is split into four parts:


Data split into 4 partitions
Fold1
Validation
Training
Training
Training
Fold2
Training
Validation
Training
Training
Fold3
Training
Training
Validation
Training
Fold4
Training
Training
Training
Validation
Average Score

One key thing to note when using the k-fold validation dataset is that it is very expensive, because you run the algorithm several times on different parts of the dataset, which can turn out to be very expensive for computation-intensive algorithms—particularly in areas of computer vision algorithms, where, sometimes, training an algorithm could take anywhere from minutes to days. So, use this technique wisely.

K-fold validation with shuffling

To make things complex and robust, you can shuffle the data every time you create your holdout validation dataset. It is very helpful for solving problems where a small boost in performance could have a huge business impact. If your case is to quickly build and deploy algorithms and you are OK with compromising a few percent in performance difference, then this approach may not be worth it. It all boils down to what problem you are trying to solve, and what accuracy means to you.

There are a few other things that you may need to consider when splitting up the data, such as:

- Data representativeness
- Time sensitivity
- Data redundancy

Data representativeness

In the example we saw in our last chapter, we classified images as either dogs or cats. Let's take a scenario where all the images are sorted and the first 60% of images are dogs and the rest are cats. If we split this dataset by choosing the first 80% as the training dataset and the rest as the validation set, then the validation dataset will not be a true representation of the dataset, as it will only contain cat images. So, in these cases, care should be taken that we have a good mix by shuffling the data before splitting or doing a stratified sampling. Stratified sampling refers to picking up data points from each category to create validation and test datasets.

Time sensitivity

Let's take the case of predicting stock prices. We have data from January to December. In this case, if we do a shuffle or stratified sampling then we end up with an information leak, as the prices could be sensitive to time. So, create the validation dataset in such a way that there is no information leak. In this case, choosing the December data as the validation dataset could make more sense. In the case of stock prices it is more complex than this, so domain-specific knowledge also comes into play when choosing the validation split.

Data redundancy

Duplicates are common in data. Care should be taken so that the data present in the training, validation, and test sets are unique. If there are duplicates, then the model may not generalize well on unseen data.

Data preprocessing and feature engineering

We have looked at different ways to split our datasets to build our evaluation strategy. In most cases, the data that we receive may not be in a format that can be readily used by us for training our algorithms. In this section, we will cover some of the preprocessing techniques and feature engineering techniques. Though most of the feature engineering techniques are domain-specific, particularly in the areas of computer vision and text, there are some common feature engineering techniques that are common across the board, which we will discuss in this chapter.

Data preprocessing for neural networks is a process in which we make the data more suitable for the deep learning algorithms to train on. The following are some of the commonly-used data preprocessing steps:

- Vectorization
- Normalization
- Missing values
- Feature extraction

Vectorization

Data comes in various formats such as text, sound, images, and video. The very first thing that needs to be done is to convert the data into PyTorch tensors. In the previous example, we used `torchvision` utility functions to convert **Python Imaging Library** (**PIL**) images into a Tensor object, though most of the complexity is abstracted away by the PyTorch torchvision libraries. In `Chapter 7`, *Generative Networks*, when we deal with **recurrent neural networks** (**RNNs**), we will see how text data can be converted into PyTorch tensors. For problems involving structured data, the data is already present in a vectorized format; all we need to do is convert them into PyTorch tensors.

Value normalization

It is a common practice to normalize features before passing the data to any machine learning algorithm or deep learning algorithm. It helps in training the algorithms faster and helps in achieving more performance. Normalization is the process in which you represent data belonging to a particular feature in such a way that its mean is zero and standard deviation is one.

In the example of *dogs and cats*, the classification that we covered in the last chapter, we normalized the data by using the mean and standard deviation of the data available in the `ImageNet` dataset. The reason we chose the `ImageNet` dataset's mean and standard deviation for our example is that we are using the weights of the ResNet model, which was pretrained on ImageNet. It is also a common practice to divide each pixel value by 255 so that all the values fall in the range between zero and one, particularly when you are not using pretrained weights.

Normalization is also applied for problems involving structured data. Say we are working on a house price prediction problem—there could be different features that could fall in different scales. For example, distance to the nearest airport and the age of the house are variables or features that could be in different scales. Using them with neural networks as they are could prevent the gradients from converging. In simple words, loss may not go down as expected. So, we should be careful to apply normalization to any kind of data before training on our algorithms. To ensure that the algorithm or model performs better, ensure that the data follows the following characteristics:

- **Take small values**: Typically in a range between zero and one
- **Same range**: Ensure all the features are in the same range

Handling missing values

Missing values are quite common in real-world machine learning problems. From our previous examples of predicting house prices, certain fields for the age of the house could be missing. It is often safe to replace the missing values with a number that may not occur otherwise. The algorithms will be able to identify the pattern. There are other techniques that are available to handle missing values that are more domain-specific.

Feature engineering

Feature engineering is the process of using domain knowledge about a particular problem to create new variables or features that can be passed to the model. To understand better, let's look at a sales prediction problem. Say we have information about promotion dates, holidays, competitor's start date, distance from competitor, and sales for a particular day. In the real world, there could be hundreds of features that may be useful in predicting the prices of stores. There could be certain information that could be important in predicting the sales. Some of the important features or derived values are:

- Days until the next promotion
- Days left before the next holiday
- Number of days the competitor's business has been open

There could be many more such features that can be extracted that come from domain knowledge. Extracting these kinds of features for any machine learning algorithm or deep learning algorithm could be quite challenging for the algorithms to perform themselves. For certain domains, particularly in the fields of computer vision and text, modern deep learning algorithms help us in getting away with feature engineering. Except for these fields, good feature engineering always helps in the following:

- The problem can be solved a lot faster with less computational resource.
- The deep learning algorithms can learn features without manually engineering them by using huge amounts of data. So, if you are tight on data, then it is good to focus on good feature engineering.

Overfitting and underfitting

Understanding overfitting and underfitting is the key to building successful machine learning and deep learning models. At the start of the chapter, we briefly covered what underfitting and overfitting are; let's take a look at them in detail and how we can solve them.

Overfitting, or not generalizing, is a common problem in machine learning and deep learning. We say a particular algorithm overfits when it performs well on the training dataset but fails to perform on unseen or validation and test datasets. This mostly occurs due to the algorithm identifying patterns that are too specific to the training dataset. In simpler words, we can say that the algorithm figures out a way to memorize the dataset so that it performs really well on the training dataset and fails to perform on the unseen data. There are different techniques that can be used to avoid the algorithm overfitting. Some of the techniques are:

- Getting more data
- Reducing the size of the network
- Applying weight regularizer
- Applying dropout

Getting more data

If you are able to get more data on which the algorithm can train, that can help the algorithm to avoid overfitting by focusing on general patterns rather than on patterns specific to small data points. There are several cases where getting more labeled data could be a challenge.

There are techniques, such as data augmentation, that can be used to generate more training data in problems related to computer vision. Data augmentation is a technique where you can adjust the images slightly by performing different actions such as rotating, cropping, and generating more data. With enough domain understanding, you can create synthetic data too if capturing actual data is expensive. There are other ways that can help to avoid overfitting when you are unable to get more data. Let's look at them.

Reducing the size of the network

The size of the network in general refers to the number of layers or the number of weight parameters used in a network. In the example of image classification that we saw in the last chapter, we used a ResNet model that has 18 blocks consisting of different layers inside it. The torchvision library in PyTorch comes with ResNet models of different sizes starting from 18 blocks and going up to 152 blocks. Say, for example, if we are using a ResNet block with 152 blocks and the model is overfitting, then we can try using a ResNet with 101 blocks or 50 blocks. In the custom architectures we build, we can simply remove some intermediate linear layers, thus preventing our PyTorch models from memorizing the training dataset. Let's look at an example code snippet that demonstrates what it means exactly to reduce the network size:

```
class Architecture1(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Architecture1, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)
        self.relu = nn.ReLU()
        self.fc3 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        return out
```

The preceding architecture has three linear layers, and let's say it overfits our training data. So, let's recreate the architecture with reduced capacity:

```
class Architecture2(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Architecture2, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```

The preceding architecture has only two linear layers, thus reducing the capacity and, in turn, potentially avoiding overfitting the training dataset.

Applying weight regularization

One of the key principles that helps to solve the problem of overfitting or generalization is building simpler models. One technique for building simpler models is to reduce the complexity of the architecture by reducing its size. The other important thing is ensuring that the weights of the network do not take larger values. Regularization provides constraints on the network by penalizing the model when the weights of the model are larger. Whenever the model uses larger weights, the regularization kicks in and increases the loss value, thus penalizing the model. There are two types of regularization possible. They are:

- **L1 regularization**: The sum of absolute values of weight coefficients are added to the cost. It is often referred to as the L1 norm of the weights.
- **L2 regularization**: The sum of squares of all weight coefficients are added to the cost. It is often referred to as the L2 norm of the weights.

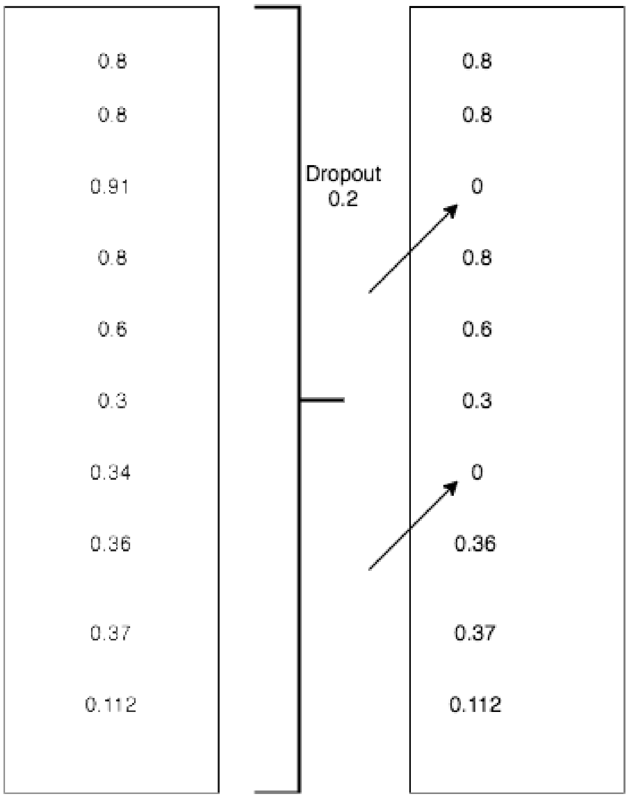
PyTorch provides an easy way to use L2 regularization by enabling the `weight_decay` parameter in the optimizer:

```
model = Architecture1(10,20,2)

optimizer = torch.optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-5)
```

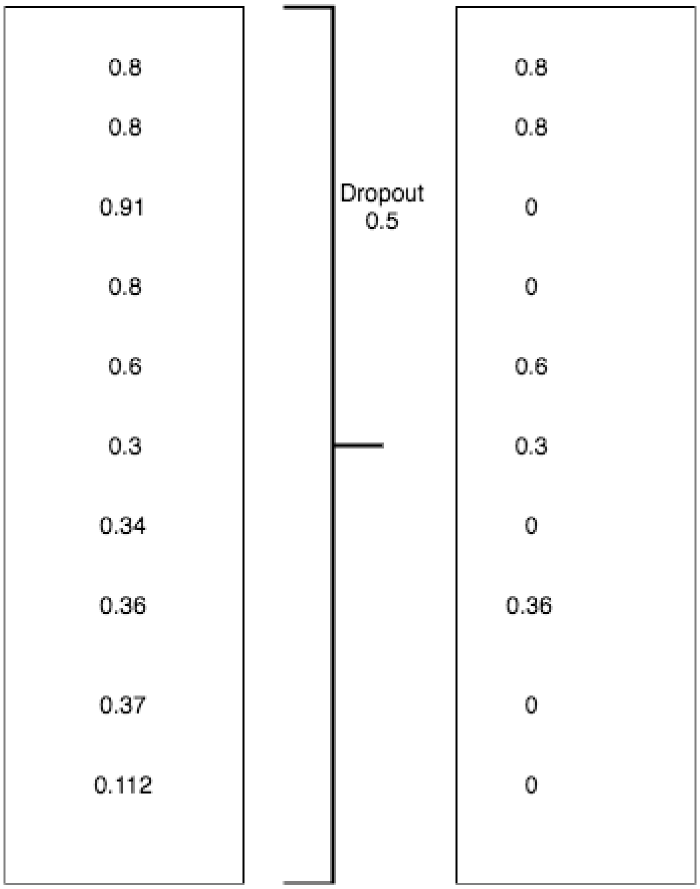
By default, the weight decay parameter is set to zero. We can try different values for weight decay; a small value such as `1e-5` works most of the time.

Dropout

Dropout is one of the most commonly used and the most powerful regularization techniques used in deep learning. It was developed by Hinton and his students at the University of Toronto. Dropout is applied to intermediate layers of the model during the training time. Let's look at an example of how dropout is applied on a linear layer's output that generates 10 values:



The preceding figure shows what happens when dropout is applied to the linear layer output with a threshold value of **0.2**. It randomly masks or zeros 20% of data, so that the model will not be dependent on a particular set of weights or patterns, thus overfitting. Let's look at another example where we apply a dropout with a threshold value of **0.5**:



It is often common to use a threshold of dropout values in the range of 0.2 to 0.5, and the dropout is applied at different layers. Dropouts are used only during the training times, and during the testing values are scaled down by the factor equal to the dropout. PyTorch provides dropout as another layer, thus making it easier to use. The following code snippet shows how to use a dropout layer in PyTorch:

```
nn.dropout(x, training=True)
```

The dropout layer accepts an argument called `training`, which needs to be set to `True` during the training phase and false during the validation or test phase.

Underfitting

There are times when our model may fail to learn any patterns from our training data, which will be quite evident when the model fails to perform well even on the dataset it is trained on. One common thing to try when your model underfits is to acquire more data for the algorithm to train on. Another approach is to increase the complexity of the model by increasing the number of layers or by increasing the number of weights or parameters used by the model. It is often a good practice not to use any of the aforementioned regularization techniques until we actually overfit the dataset.

Workflow of a machine learning project

In this section, we will formalize a solution framework that can be used to solve any machine learning problem by bringing together the problem statement, evaluation, feature engineering, and avoidance of overfitting.

Problem definition and dataset creation

To define the problem, we need two important things; namely, the input data and the type of problem.

What will be our input data and target labels? For example, say we want to classify restaurants based on their speciality—say Italian, Mexican, Chinese, and Indian food—from the reviews given by the customers. To start working with this kind of problem, we need to manually hand annotate the training data as one of the possible categories before we can train the algorithm on it. Data availability is often a challenging factor at this stage.

Identifying the type of problem will help in deciding whether it is a binary classification, multi-classification, scalar regression (house pricing), or vector regression (bounding boxes). Sometimes, we may have to use some of the unsupervised techniques such as clustering and dimensionality reduction. Once the problem type is identified, then it becomes easier to determine what kind of architecture, loss function, and optimizer should be used.

Once we have the inputs and have identified the type of the problem, then we can start building our models with the following assumptions:

- There are hidden patterns in the data that can help map the input with the output

- The data that we have is sufficient for the model to learn

As machine learning practitioners, we need to understand that we may not be able to build a model with just some input data and target data. Let's take predicting stock prices as an example. Let's assume we have features representing historical prices, historical performance, and competition details, but we may still fail to build a meaningful model that can predict stock prices, as stock prices could actually be influenced by a variety of other factors such as the domestic political scenario, international political scenario, natural factors such as having a good monsoon, and many other factors that may not be represented by our input data. So, there is no way that any machine learning or deep learning model would be able to identify patterns. So, based on the domain, carefully pick features that can be real indicators of the target variable. All these could be reasons for the models to underfit.

There is another important assumption that machine learning makes. Future or unseen data will be close to the patterns, as described by the historical data. Sometimes, our models could fail, as the patterns never existed in the historical data, or the data on which the model was trained did not cover certain seasonalities or patterns.

Measure of success

The measure of success will be directly determined by your business goal. For example, when trying to predict when the next machine failure will occur in windmills, we would be more interested to know how many times the model was able to predict the failures. Using simple accuracy can be the wrong metric, as most of the time the model will predict correctly when the machine will not fail, as that is the most common output. Say we get an accuracy of 98%, and the model was wrong each time in predicting the failure rate—such models may not be of any use in the real world. Choosing the correct measure of success is crucial for business problems. Often, these kinds of problems have imbalanced datasets.

For balanced classification problems, where all the classes have a likely accuracy, ROC and **Area under the curve** (**AUC**) are common metrics. For imbalanced datasets, we can use precision and recall. For ranking problems, we can use mean average precision.

Evaluation protocol

Once you decide how you are going to evaluate the current progress, it is important to decide how you are going to evaluate on your dataset. We can choose from the three different ways of evaluating our progress:

- **Holdout validation set**: Most commonly used, particularly when you have enough data
- **K-fold cross validation**: When you have limited data, this strategy helps you to evaluate on different portions of the data, helping to give us a better view of the performance
- **Iterated k-fold validation**: When you are looking to go the extra mile with the performance of the model, this approach will help

Prepare your data

Bring different formats of available data into tensors through vectorization and ensure that all the features are scaled and normalized.

Baseline model

Create a very simple model that beats the baseline score. In our previous example of dogs and cats, classification, the baseline accuracy should be 0.5 and our simple model should be able to beat this score. If we are not able to beat the baseline score, then maybe the input data does not hold the necessary information required to make the necessary prediction. Remember not to introduce any regularization or dropouts at this step.

To make the model work, we have to make three important choices:

- **Choice of last layer**: For a regression, it should be a linear layer generating a scalar value as output. For a vector regression problem, it would be the same linear layer generating more than one scalar output. For a bounding box, it outputs four values. For a binary classification, it is often common to use sigmoid, and for multi-class classification it is softmax.
- **Choice of loss function**: The type of the problem will help you in deciding the loss function. For a regression problem, such as predicting house prices, we use the mean squared error, and for classification problems we use categorical cross entropy.
- **Optimization**: Choosing the right optimization algorithm and some of its hyper parameters is quite tricky, and we can find

> them by experimenting with different ones. For most of the use cases, an Adam or RMSprop optimization algorithm works better. We will cover some of the tricks that can be used for learning rate selection.

Let's summarize what kind of loss function and activation function we would use for the last layer of the network in our deep learning algorithms:

Problem type	**Activation function**	**Loss function**
Binary classification	Sigmoid activation	`nn.CrossEntropyLoss()`
Multi-class classification	Softmax activation	`nn.CrossEntropyLoss()`
Multi-label classification	Sigmoid activation	`nn.CrossEntropyLoss()`
Regression	None	MSE
Vector regression	None	MSE

Large model enough to overfit

Once you have a model that has enough capacity to beat your baseline score, increase your baseline capacity. A few simple tricks to increase the capacity of your architecture are as follows:

- Add more layers to your existing architecture
- Add more weights to the existing layers
- Train it for more epochs

We generally train the model for an adequate number of epochs. Stop it when the training accuracy keeps increasing and the validation accuracy stops increasing and probably starts dropping; that's where the model starts overfitting. Once we reach this stage, we need to apply regularization techniques.

Remember, the number of layers, size of layers, and number of epochs may change from problem to problem. A smaller architecture can work for a simple classification problem, but for a complex problem such as facial recognition, we would need enough expressiveness in our architecture and the model needs to be trained for more epochs than for a simple classification problem.

Applying regularization

Finding the best way to regularize the model or algorithm is one of the trickiest parts of the process, since there are a lot of parameters to be tuned. Some of the parameters that we can tune to regularize the model are:

- **Adding dropout**: This can be complex as this can be added between different layers, and finding the best place is usually done through experimentation. The percentage of dropout to be added is also tricky, as it is purely dependent on the problem statement we are trying to solve. It is often good practice to start with a small number such as 0.2.
- **Trying different architectures**: We can try different architectures, activation functions, numbers of layers, weights, or parameters inside the layers.
- **Adding L1 or L2 regularization**: We can make use of either one of regularization.
- **Trying different learning rates**: There are different techniques that can be used, which we will discuss in the later sections of the chapter.
- **Adding more features or more data**: This is probably done by acquiring more data or augmenting data.

We will use our validation dataset to tune all the aforementioned hyper parameters. As we keep iterating and tweaking the hyper parameters, we may end up with the problem of data leakage. So, we should ensure that we have holdout data for testing. If the performance of the model on the test data is good in comparison to the training and validation, then there is a good chance that our model will perform well on unseen data. But, if the model fails to perform on the test data but performs on the validation and training data, then there is a chance that the validation data is not a good representation of the real-world dataset. In such scenarios, we can end up using k-fold validation or iterated k-fold validation datasets.

Learning rate picking strategies

Finding the right learning rate for training the model is an ongoing area of research where a lot of progress has been made. PyTorch provides some of the techniques to tune the learning rate, and they are provided in the `torch.optim.lr_sheduler` package. We will explore some of the techniques that PyTorch provides to choose the learning rates dynamically:

- **StepLR**: This scheduler takes two important parameters. One is step size, which denotes for what number of epochs the learning rate has to change, and the second parameter is gamma, which decides how much the learning rate has to be changed.

For a learning rate of `0.01`, step size of 10, and gamma size of `0.1`, for every 10 epochs the learning rate changes by gamma times. That is, for the first 10 epochs, the learning rate changes to 0.001, and by the end, on the next 10 epochs, it changes to 0.0001. The following code explains the implementation of `StepLR`:

```
scheduler = StepLR(optimizer, step_size=30, gamma=0.1)
for epoch in range(100):
    scheduler.step()
    train(...)
    validate(...)
```

- **MultiStepLR**: MultiStepLR works similarly to StepLR, except for the fact that the steps are not at regular intervals; steps are given as lists. For example, it is given as a list of 10, 15, 30, and for each step value, the learning rate is multiplied by its gamma value. The following code explains the implementation of `MultiStepLR`:

```
scheduler = MultiStepLR(optimizer, milestones=[30,80], gamma=0.1)
for epoch in range(100):
    scheduler.step()
    train(...)
    validate(...)
```

- **ExponentialLR**: This sets the learning rate to a multiple of the learning rate with gamma values for each epoch.
- **ReduceLROnPlateau**: This is one of the commonly used learning rate strategies. In this case, the learning rate changes when a particular metric, such as training loss, validation loss, or accuracy stagnates. It is a common practice to reduce the learning rate by two to 10 times its original value. `ReduceLROnPlateau` can be implemented as follows:

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.1,
  momentum=0.9)
scheduler = ReduceLROnPlateau(optimizer, 'min')
for epoch in range(10):
    train(...)
    val_loss = validate(...)
```

```
            # Note that step should be called after validate()
            scheduler.step(val_loss)
```

Summary

In this chapter, we covered some of the common and best practices that are used in solving machine learning or deep learning problems. We covered various important steps such as creating problem statements, choosing the algorithm, beating the baseline score, increasing the capacity of the model until it overfits the dataset, applying regularization techniques that can prevent overfitting, increasing the generalization capacity, tuning different parameters of the model or algorithms, and exploring different learning strategies that can be used to train deep learning models optimally and faster.

In the next chapter, we will cover different components that are responsible for building state-of-the-art **Convolutional Neural Networks** (**CNNs**). We will also cover transfer learning, which helps us to train image classifiers when little data is available. We will also cover techniques that help us to train these algorithms more quickly.

Deep Learning for Computer Vision

In `Chapter 3`, *Diving Deep into Neural Networks,* we built an image classifier using a popular **Convolutional Neural Network** (**CNN**) architecture called **ResNet**, but we used this model as a black box. In this chapter, we will cover the important building blocks of convolutional networks. Some of the important topics that we will be covering in this chapter are:

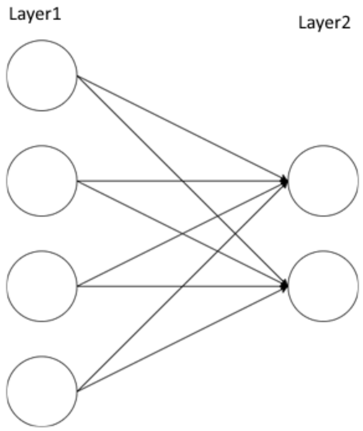
- Introduction to neural networks
- Building a CNN model from scratch
- Creating and exploring a VGG16 model
- Calculating pre-convoluted features
- Understanding what a CNN model learns
- Visualizing weights of the CNN layer

We will explore how we can build an architecture from scratch for solving image classification problems, which are the most common use cases. We will also learn how to use transfer learning, which will help us in building image classifiers using a very small dataset.

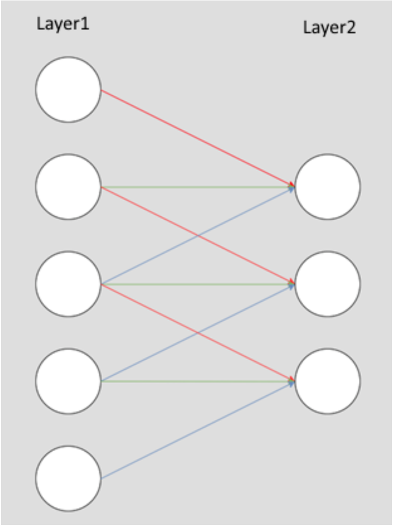
Apart from learning how to use CNNs, we will also explore what these convolutional networks learn.

Introduction to neural networks

In the last few years, CNNs have become popular in the areas of image recognition, object detection, segmentation, and many other tasks in the field of computer vision. They are also becoming popular in the field of **natural language processing** (**NLP**), though they are not commonly used yet. The fundamental difference between fully connected layers and convolution layers is the way the weights are connected to each other in the intermediate layers. Let's take a look at an image where we depict how fully connected, or linear, layers work:



One of the biggest challenges of using a linear layer or fully connected layers for computer vision is that they lose all spatial information, and the complexity in terms of the number of weights used by fully connected layers is too big. For example, when we represent a 224 pixel image as a flat array, we would end up with 150, 528 (224 x 224 x 3 channels). When the image is flattened, we lose all the spatial information. Let's look at what a simplified version of a CNN looks like:



All the convolution layer is doing is applying a window of weights called **filters** across the image. Before we try to understand convolutions and other building blocks in detail, let's build a simple yet powerful image classifier for the `MNIST` dataset. Once we build this, we will walk through each component of the network. We will break down building our image classifier into the following steps:

- Getting data
- Creating a validation dataset
- Building our CNN model from scratch
- Training and validating the model

MNIST – getting data

The `MNIST` dataset contains 60,000 handwritten digits from 0 to 9 for training, and 10,000 images for a test set. The PyTorch `torchvision` library provides us with an `MNIST` dataset, which downloads the data and provides it in a readily-usable format. Let's use the dataset `MNIST` function to pull the dataset to our local machine, and then wrap it around a `DataLoader`. We will use torchvision transformations to convert the data into PyTorch tensors and do data normalization. The following code takes care of downloading, wrapping around the `DataLoader` and normalizing the data:

```
transformation =
  transforms.Compose([transforms.ToTensor(),
  transforms.Normalize((0.1307,), (0.3081,))])

train_dataset =
  datasets.MNIST('data/',train=True,transform=transformation,
    download=True)
test_dataset =
  datasets.MNIST('data/',train=False,transform=transformation,
    download=True)

train_loader =
  torch.utils.data.DataLoader(train_dataset,batch_size=32,shuffle=True)
test_loader =
  torch.utils.data.DataLoader(test_dataset,batch_size=32,shuffle=True)
```

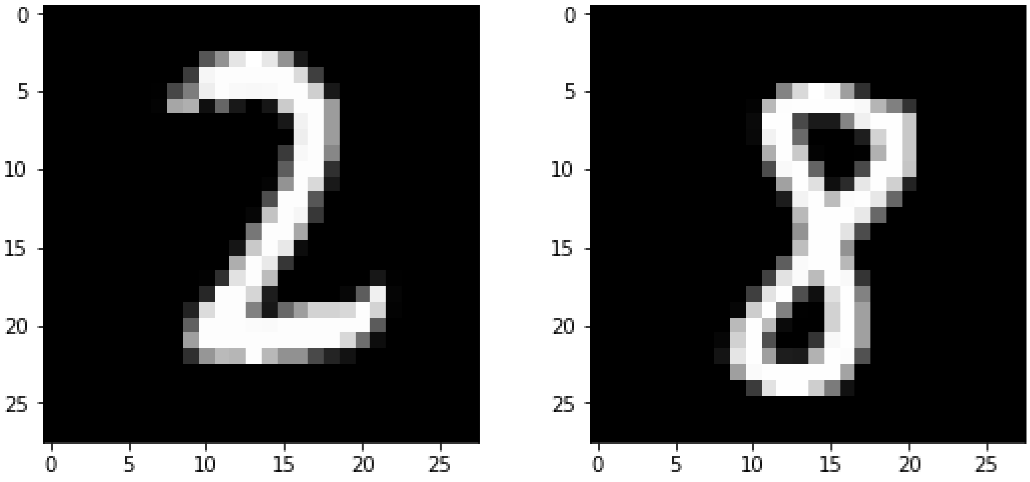
So, the previous code provides us with a `DataLoader` for the `train` and `test` datasets. Let's visualize a few images to get an understanding of what we are dealing with. The following code will help us in visualizing the MNIST images:

```
def plot_img(image):
    image = image.numpy()[0]
    mean = 0.1307
    std = 0.3081
    image = ((mean * image) + std)
    plt.imshow(image,cmap='gray')
```

Now we can pass the `plot_img` method to visualize our dataset. We will pull a batch of records from the `DataLoader` using the following code, and plot the images:

```
sample_data = next(iter(train_loader))
plot_img(sample_data[0][1])
plot_img(sample_data[0][2])
```

The images are visualized as follows:

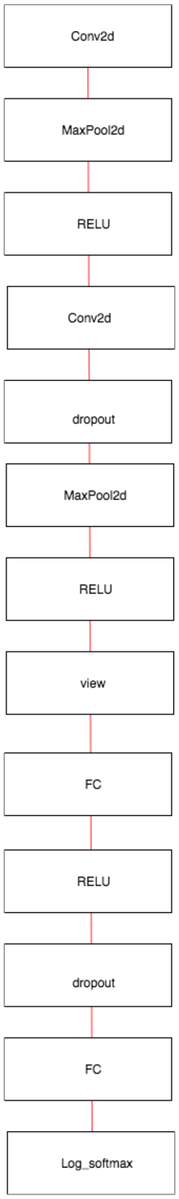


Building a CNN model from scratch

For this example, let's build our own architecture from scratch. Our network architecture will contain a combination of different layers, namely:

- `Conv2d`
- `MaxPool2d`
- **Rectified linear unit** (**ReLU**)
- View
- Linear layer

Let's look at a pictorial representation of the architecture we are going to implement:


Conv2d
MaxPool2d
RELU
Conv2d
dropout
MaxPool2d
RELU
view
FC
RELU
dropout
FC
Log_softmax

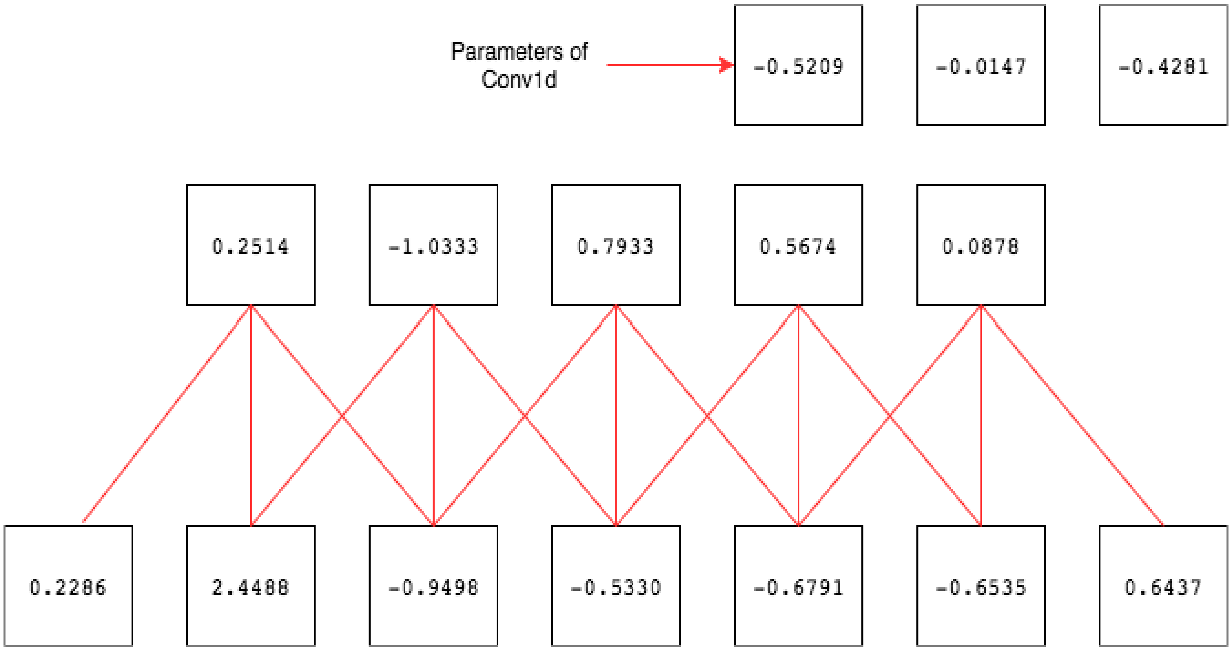
Let's implement this architecture in PyTorch and then walk through what each individual layer does:

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)
```

Let's understand in detail what each layer does.

Conv2d

`Conv2d` takes care of applying a convolutional filter on our MNIST images. Let's try to understand how convolution is applied on a one-dimensional array, and then move to how a two-dimensional convolution is applied to an image. We will look at the following image, to which we will apply a `Conv1d` of a filter (or kernel) size `3` to a tensor of length `7`:


Parameters of Conv1d
-0.5209
-0.0147
-0.4281
0.2514
-1.0333
0.7933
0.5674
0.0878
0.2286
2.4488
-0.9498
-0.5330
-0.6791
-0.6535
0.6437

The bottom boxes represent our input tensor of seven values, and the connected boxes represent the output after we apply our convolution filter of size three. At the top-right corner of the image, the three boxes represent the weights and parameters of the `Conv1d` layer. The convolution filter is applied like a window and it moves to the next values by skipping one value. The number of values to be skipped is called the **stride,** and is set to `1` by default. Let's understand how the output values are being calculated by writing down the calculation for the first and last outputs:

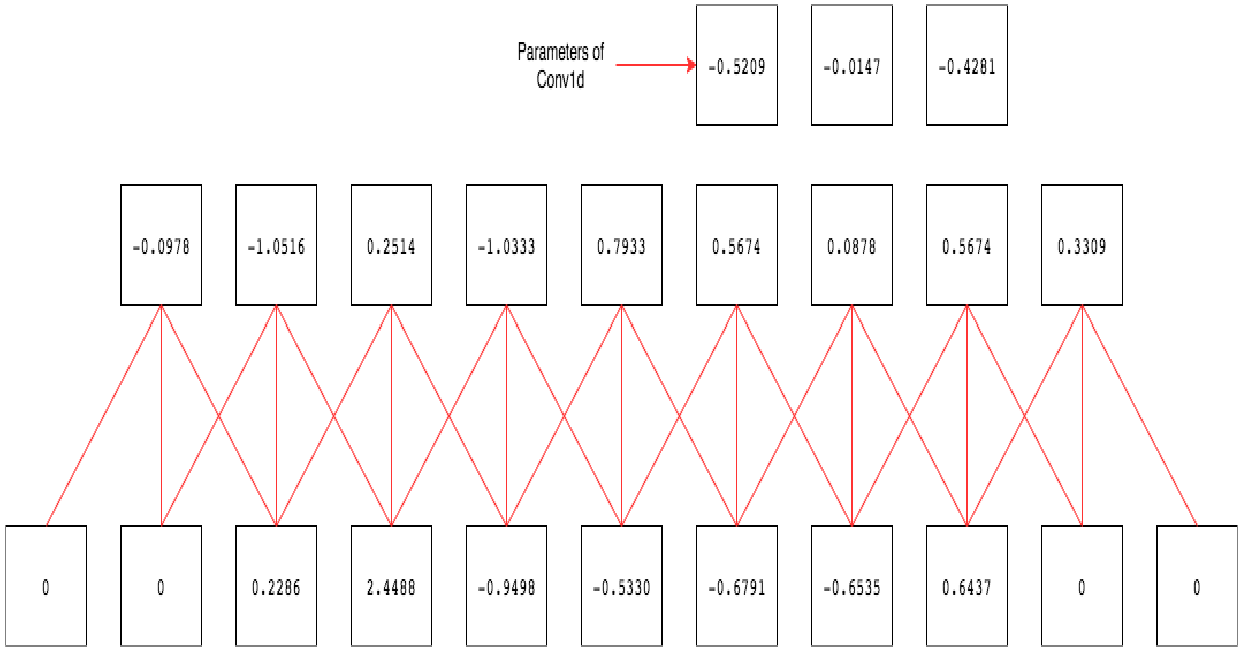
Output 1 –> (*-0.5209 x 0.2286*) + (*-0.0147 x 2.4488*) + (*-0.4281 x -0.9498*)

Output 5 –> (*-0.5209 x -0.6791*) + (*-0.0147 x -0.6535*) + (*-0.4281 x 0.6437*)

So, by now, it should be clear what a convolution does. It applies a filter (or kernel), which is a bunch of weights, on the input by moving it based on the value of the stride. In the previous example, we are moving our filter one at a time. If the stride value is `2`, then we would move two points at a time. Let's look at a PyTorch implementation to understand how it works:

```
conv = nn.Conv1d(1,1,3,bias=False)
sample = torch.randn(1,1,7)
conv(Variable(sample))

#Check the weights of our convolution filter by
conv.weight
```

There is another important parameter, called **padding,** which is often used with convolutions. If we keenly observe the previous example, we may realize that if the filter is not applied until the end of the data, when there are not enough elements for the data to stride, it stops. Padding prevents this by adding zeros to both ends of a tensor. Let's again look at a one-dimensional example of how padding works:


Parameters of Conv1d
-0.5209
-0.0147
-0.4281
-0.0978
-1.0516
0.2514
-1.0333
0.7933
0.5674
0.0878
0.5674
0.3309
0
0
0.2286
2.4488
-0.9498
-0.5330
-0.6791
-0.6535
0.6437
0
0

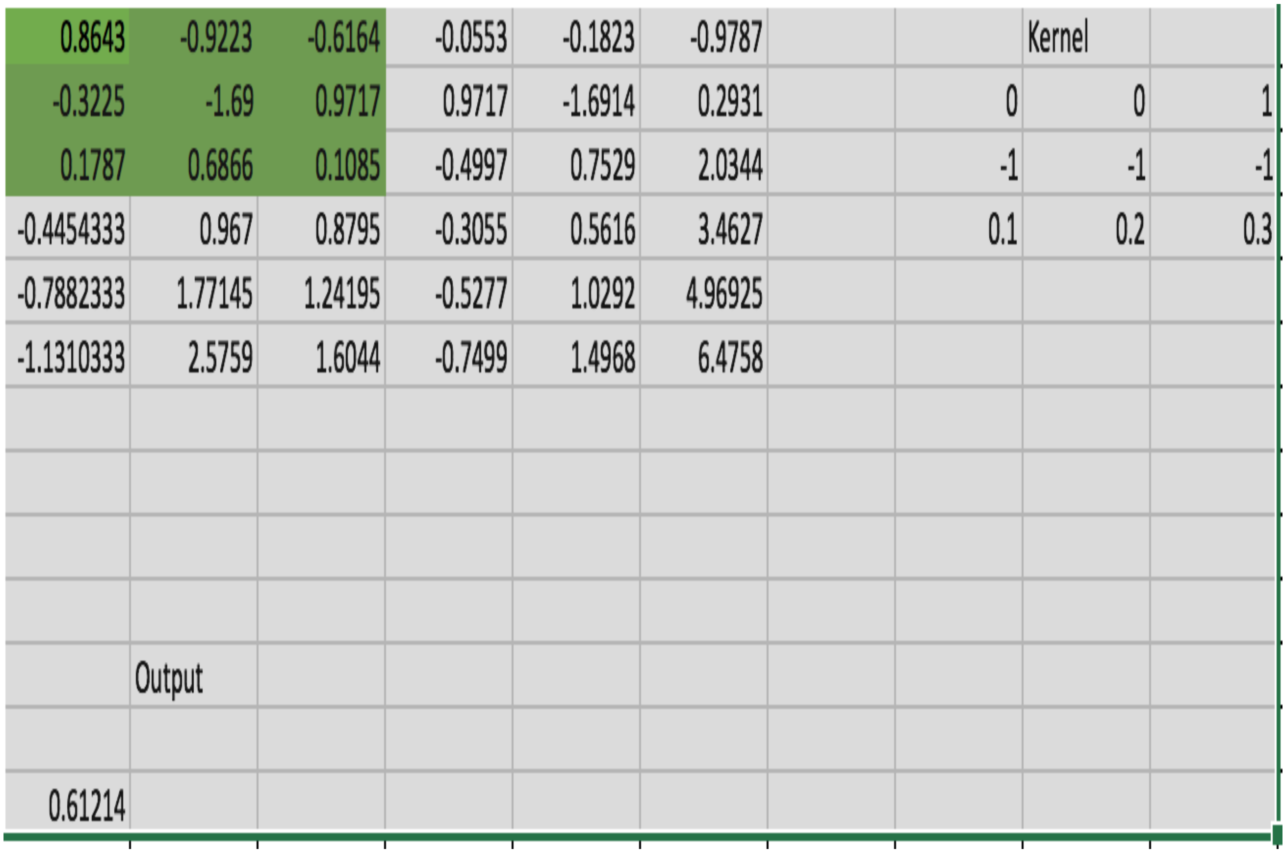
In the preceding image, we applied a `Conv1d` layer with padding `2` and stride `1`.

Let's look at how Conv2d works on an image:

Before we understand how Conv2d works, I would strongly recommend you to check the amazing blog (`http://setosa.io/ev/image-kernels/`) where it contains a live demo of how a convolution works. After you have spent few minutes playing with the demo, read the next section.

Let's understand what happened in the demo. In the center box of the image, we have two different sets of numbers; one represented in the boxes and the other beneath the boxes. The ones represented in the boxes are pixel values, as highlighted by the white box on the left-hand photo. The numbers denoted beneath the boxes are the filter (or kernel) values that are being used to sharpen the image. The numbers are handpicked to do a particular job. In this case, it is sharpening the image. Just like in our previous example, we are doing an element-to-element multiplication and summing up all the values to generate the value of the pixel in the right-hand image. The generated value is highlighted by the white box on the right-hand side of the image.

Though the values in the kernel are handpicked in this example, in CNNs we do not handpick the values, but rather we initialize them randomly and let the gradient descent and backpropagation tune the values of the kernels. The learned kernels will be responsible for identifying different features such as lines, curves, and eyes. Let's look at another image where we look at it as a matrix of numbers and see how convolution works:



0.8643	-0.9223	-0.6164	-0.0553	-0.1823	-0.9787			Kernel	
-0.3225	-1.69	0.9717	0.9717	-1.6914	0.2931		0	0	1
0.1787	0.6866	0.1085	-0.4997	0.7529	2.0344		-1	-1	-1
-0.4454333	0.967	0.8795	-0.3055	0.5616	3.4627		0.1	0.2	0.3
-0.7882333	1.77145	1.24195	-0.5277	1.0292	4.96925				
-1.1310333	2.5759	1.6044	-0.7499	1.4968	6.4758				
	Output								
0.61214									

In the preceding screenshot, we assume that the 6 x 6 matrix represents an image and we apply the convolution filter of size 3 x 3, then we show how the output is generated. To keep it simple, we are just calculating for the highlighted portion of the matrix. The output is generated by doing the following calculation:

Output –> *0.86 x 0 + -0.92 x 0 + -0.61 x 1 + -0.32 x -1 + -1.69 x -1 +* ....

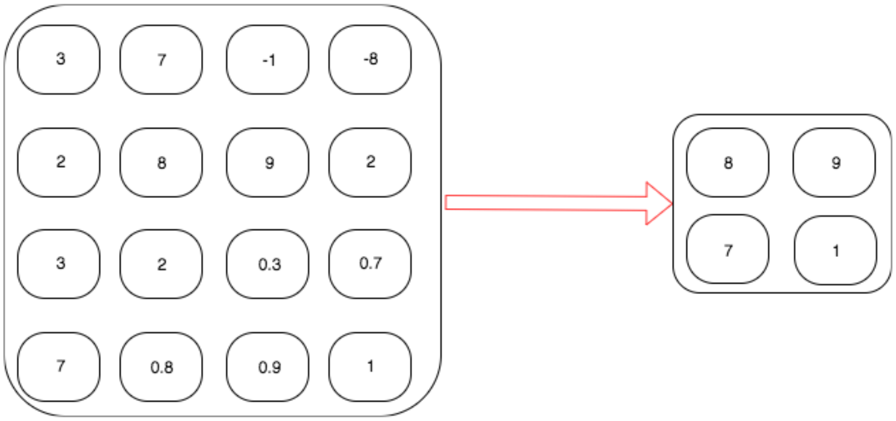
The other important parameter used in the `Conv2d` function is `kernel_size`, which decides the size of the kernel. Some of the commonly used kernel sizes are *1*, *3*, *5*, and *7*. The larger the kernel size, the larger the area that a filter can cover becomes huge, so it is common to observe filters of *7* or *9* being applied to the input data in the early layers.

Pooling

It is a common practice to add pooling layers after convolution layers, as they reduce the size of feature maps and the outcomes of convolution layers.

Pooling offers two different features: one is reducing the size of data to process, and the other is forcing the algorithm to not focus on small changes in the position of an image. For example, a face-detecting algorithm should be able to detect a face in the picture, irrespective of the position of the face in the photo.

Let's look at how MaxPool2d works. It also has the same concept of kernel size and strides. It differs from convolutions as it does not have any weights, and just acts on the data generated by each filter from the previous layer. If the kernel size is *2 x 2,* then it considers that size in the image and picks the max of that area. Let's look at the following image, which will make it clear how MaxPool2d works:



3	7	-1	-8
2	8	9	2
3	2	0.3	0.7
7	0.8	0.9	1

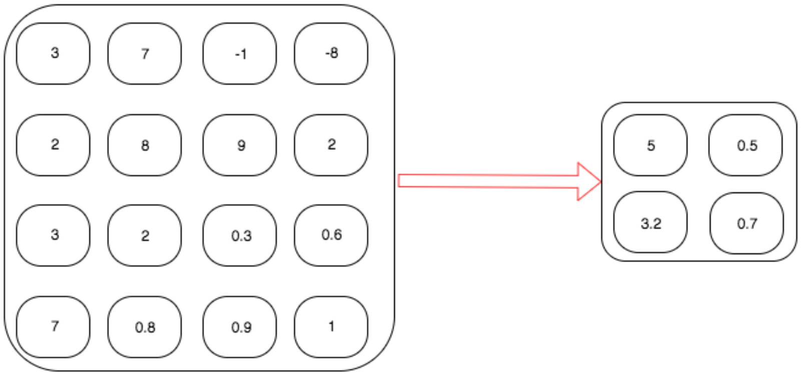
8	9
7	1

The box on the left-hand side contains the values of feature maps. After applying max-pooling, the output is stored on the right-hand side of the box. Let's look at how the output is calculated, by writing down the calculation for the values in the first row of the output:

$$Output1-> Maximum(3,7,2,8)- > 8$$

$$Output2-> Maximum(-1,-8,9,2)- > 9$$

The other commonly used pooling technique is **average pooling**. The `maximum` function is replaced by the `average` function. The following image explains how average pooling works:



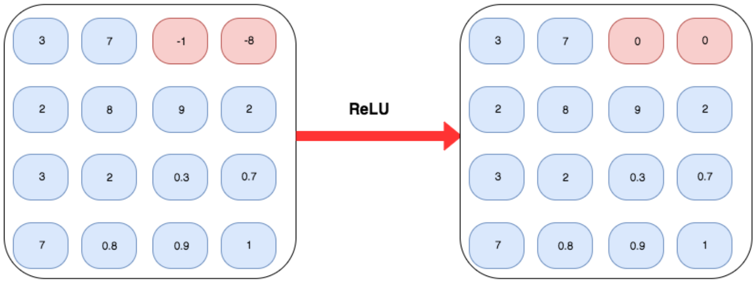
In this example, instead of taking a maximum of four values, we are taking the average four values. Let's write down the calculation to make it easier to understand:

$$Output1-> Average(3,7,2,8)- > 84$$

$$Output2-> Average(-1,-8,9,2)- > -37$$

Nonlinear activation – ReLU

It is a common and a best practice to have a nonlinear layer after max-pooling, or after convolution is applied. Most of the network architectures tend to use ReLU or different flavors of ReLU. Whatever nonlinear function we choose, it gets applied to each element of the feature maps. To make it more intuitive, let's look at an example where we apply ReLU for the same feature map to which we applied max-pooling and average pooling:

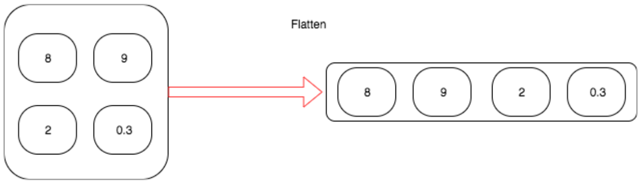


3	7	-1	-8
2	8	9	2
3	2	0.3	0.7
7	0.8	0.9	1

3	7	0	0
2	8	9	2
3	2	0.3	0.7
7	0.8	0.9	1

View

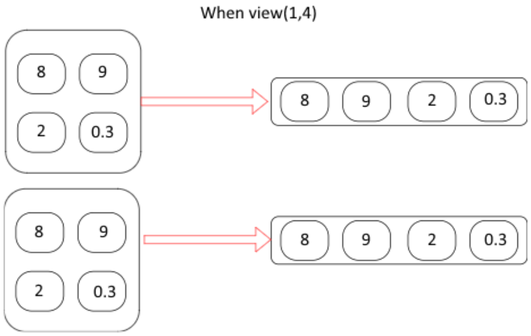
It is a common practice to use a fully connected, or linear, layer at the end of most networks for an image classification problem. We are using a two-dimensional convolution that takes a matrix of numbers as input and outputs another matrix of numbers. To apply a linear layer, we need to flatten the matrix which is a tensor of two-dimensions to a vector of one-dimension. The following example will show you how `view` works:



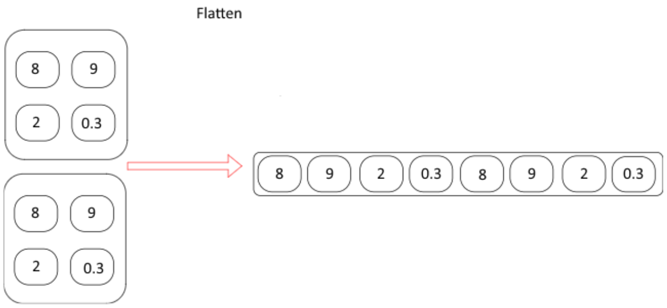
Let's look at the code used in our network that does the same:

```
x.view(-1, 320)
```

As we saw earlier, the `view` method will flatten an *n*-dimension tensor to a one-dimensional tensor. In our network, the first dimension is of each image. The input data after batching will have a dimension of *32 x 1 x 28 x 28,* where the first number, *32,* will denote that there are *32* images of size *28* height, *28* width, and *1* channel since it is a black-and-white image. When we flatten, we do not want to flatten or mix the data for different images. So, the first argument that we pass to the `view` function will instruct PyTorch to avoid flattening the data on the first dimension. Let's take a look at how this works in the following image:


When view(1,4)
8
9
2
0.3
8
9
2
0.3
8
9
2
0.3
8
9
2
0.3

In the preceding example, we have data of size *2 x 1 x 2 x 2*; after we apply the `view` function, it converts to a tensor of size *2 x 1 x 4*. Let's also look at another example where we don't mention the - *1*:


Flatten
8
9
2
0.3
8
9
2
0.3
8
9
2
0.3
8
9
2
0.3

If we ever forget to mention which dimension to flatten, we may end up with unexpected results. So be extra careful at this step.

Linear layer

After we convert the data from a two-dimensional tensor to a one-dimensional tensor, we pass the data through a linear layer, followed by a nonlinear activation layer. In our architecture, we have two linear layers; one followed by ReLU, and the other followed by a `log_softmax`, which predicts what digit is contained in the given image.

Training the model

Training the model is the same process as for our previous dogs and cats image classification problems. The following code snippet does the training of our model on the provided dataset:

```
def fit(epoch,model,data_loader,phase='training',volatile=False):
    if phase == 'training':
        model.train()
    if phase == 'validation':
        model.eval()
        volatile=True
    running_loss = 0.0
    running_correct = 0
    for batch_idx , (data,target) in enumerate(data_loader):
        if is_cuda:
            data,target = data.cuda(),target.cuda()
        data , target = Variable(data,volatile),Variable(target)
        if phase == 'training':
            optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output,target)

        running_loss += F.nll_loss(output,target,size_average=False).data[0]
        preds = output.data.max(dim=1,keepdim=True)[1]
        running_correct += preds.eq(target.data.view_as(preds)).cpu().sum()
        if phase == 'training':
            loss.backward()
            optimizer.step()

    loss = running_loss/len(data_loader.dataset)
    accuracy = 100. * running_correct/len(data_loader.dataset)

    print(f'{phase} loss is {loss:{5}.{2}} and {phase} accuracy is {running_c
    return loss,accuracy
```

This method has different logic for `training` and `validation`. There are primarily two reasons for using different modes:

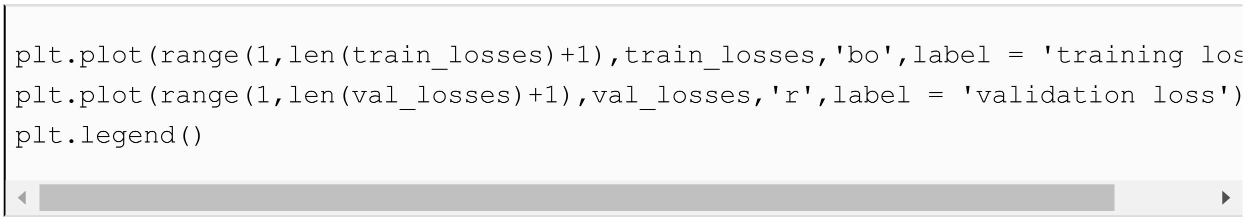
- In `train` mode, dropout removes a percentage of values, which should not happen in the validation or testing phase
- For `training` mode, we calculate gradients and change the model's parameters value, but backpropagation is not required during the testing or validation phases

Most of the code in the previous function is self-explanatory, as discussed in previous chapters. At the end of the function, we return the `loss` and `accuracy` of the model for that particular epoch.

Let's run the model through the preceding function for 20 iterations and plot the `loss` and `accuracy` of `train` and `validation`, to understand how our network performed. The following code runs the `fit` method for the `train` and `test` dataset for `20` iterations:

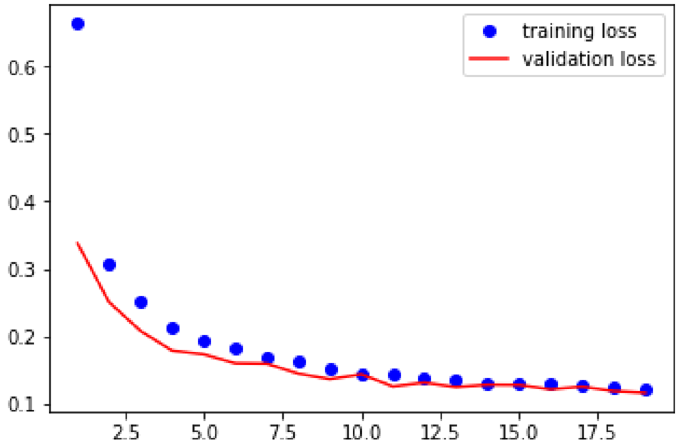
```
model = Net()
if is_cuda:
    model.cuda()

optimizer = optim.SGD(model.parameters(),lr=0.01,momentum=0.5)
train_losses , train_accuracy = [],[]
val_losses , val_accuracy = [],[]
for epoch in range(1,20):
    epoch_loss, epoch_accuracy = fit(epoch,model,train_loader,phase='training
    val_epoch_loss , val_epoch_accuracy = fit(epoch,model,test_loader,phase='
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    val_losses.append(val_epoch_loss)
    val_accuracy.append(val_epoch_accuracy)
```

The following code plots the `training` and `test loss`:

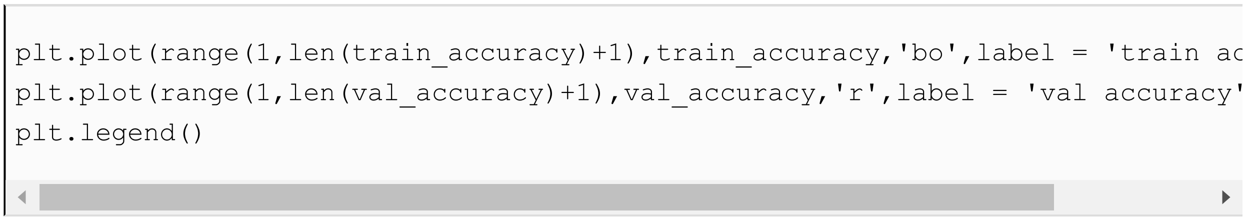


```
plt.plot(range(1,len(train_losses)+1),train_losses,'bo',label = 'training los
plt.plot(range(1,len(val_losses)+1),val_losses,'r',label = 'validation loss')
plt.legend()
```

The preceding code generates the following graph:

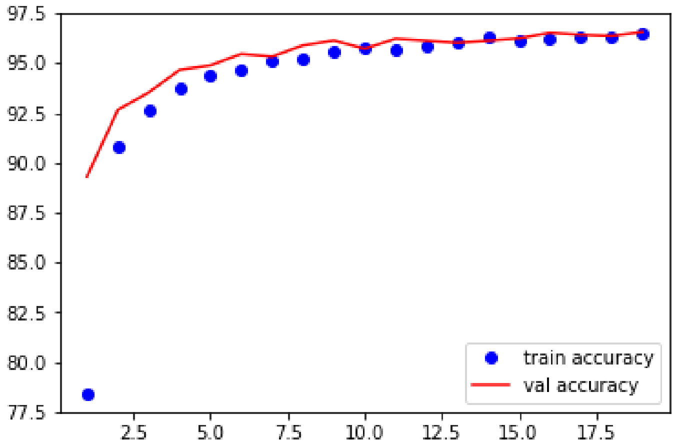


The following code plots the training and test accuracy:



```
plt.plot(range(1,len(train_accuracy)+1),train_accuracy,'bo',label = 'train ac
plt.plot(range(1,len(val_accuracy)+1),val_accuracy,'r',label = 'val accuracy'
plt.legend()
```

The preceding code generates the following graph:



At the end of the 20th epoch, we achieve a `test` accuracy of 98.9%. We have got our simple convolutional model working and almost achieving state-of-the-art results. Let's take a look at what happens when we try the same network architecture on the previously-used `Dogs vs. Cats` dataset. We will use the data from our previous chapter, `Chapter 3`, *Building Blocks of Neural Networks,* and architecture from the MNIST example with some minor changes. Once we train the model, let's evaluate it to understand how well our simple architecture performs.

Classifying dogs and cats – CNN from scratch

We will use the same architecture with a few minor changes, as listed here:

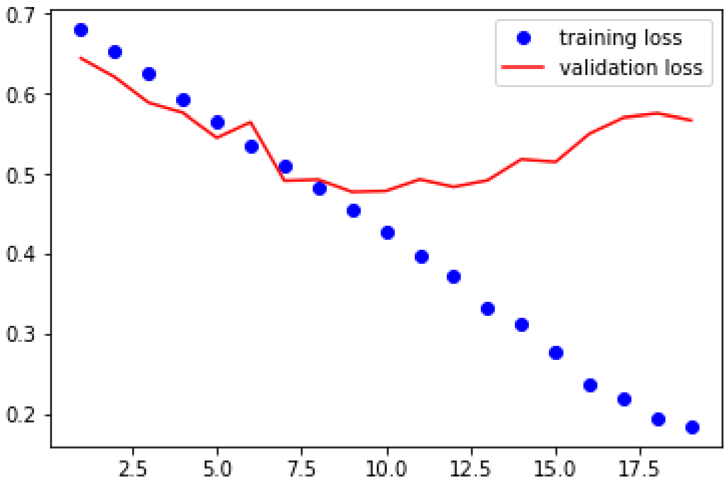
- The input dimensions for the first linear layer changes, as the dimensions for our cats and dogs images are *256, 256*
- We add another linear layer to give more flexibility for the model to learn

Let's look at the code that implements the network architecture:

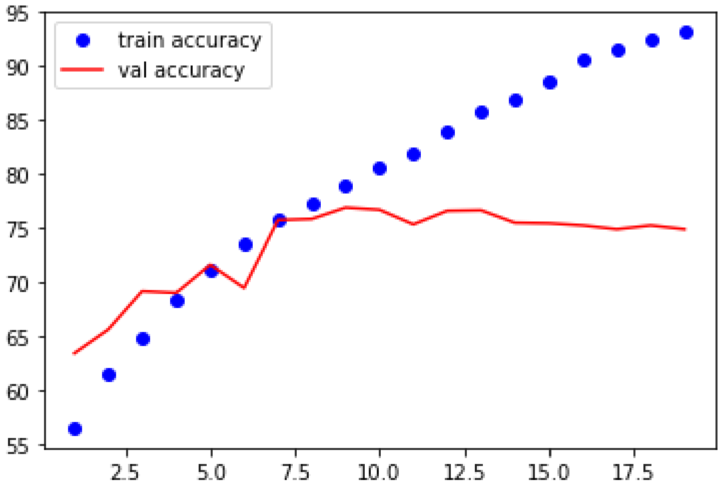
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(56180, 500)
        self.fc2 = nn.Linear(500,50)
        self.fc3 = nn.Linear(50, 2)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(x.size(0),-1)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = F.relu(self.fc2(x))
        x = F.dropout(x,training=self.training)
        x = self.fc3(x)
        return F.log_softmax(x,dim=1)
```

We will use the same `training` function which was used for the MNIST example. So, I am not including the code here. But let's look at the plots generated when the model is trained for *20* iterations.

Loss of `training` and `validation` datasets:



Accuracy for `training` and `validation` datasets:

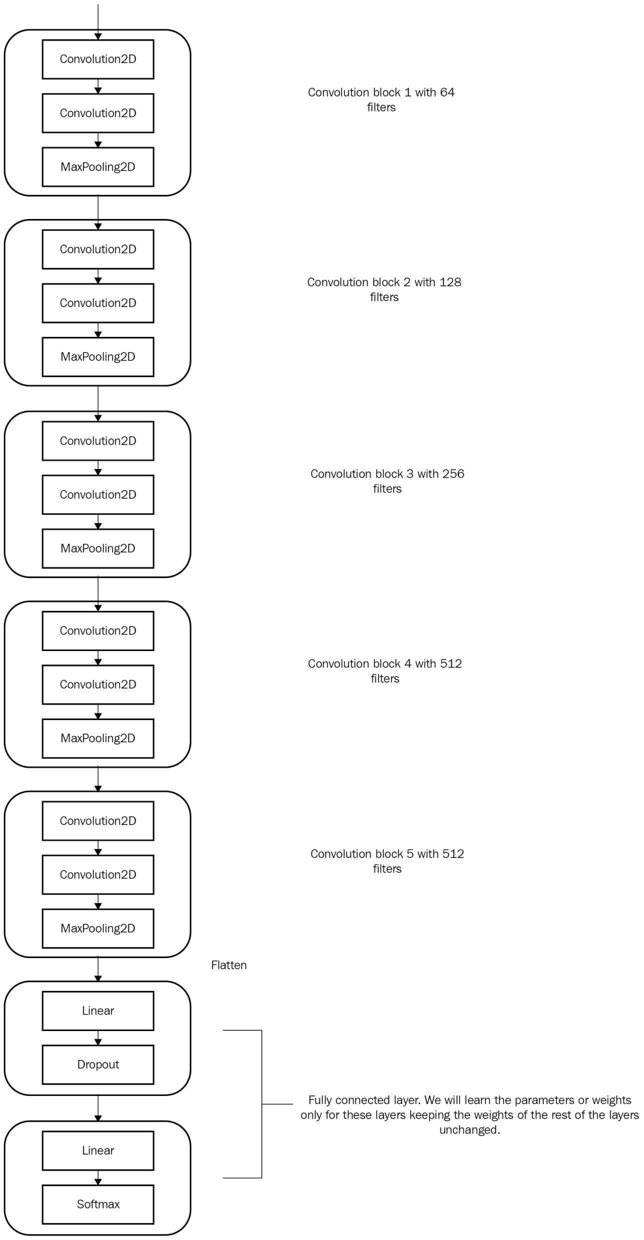


It's clear from the plots that the training loss is decreasing for every iteration, but the validation loss gets worse. Accuracy also increases during the training, but almost saturates at 75%. It is a clear example where the model is not generalizing. We will look at another technique called **transfer learning**, which helps us in training more accurate models, along with providing tricks to make the training faster.

Classifying dogs and cats using transfer learning

Transfer learning is the ability to reuse a trained algorithm on a similar dataset without training it from scratch. We humans do not learn to recognize new images by analyzing thousands of similar images. We, as humans, just understand the different features that actually differentiate a particular animal, say a fox from a dog. We do not need to learn what a fox is from understanding what lines, eyes, and other smaller features are like. So we will learn how to use a pre-trained model to build state-of-the-art image classifiers with very little data.

The first few layers of a CNN architecture focus on smaller features, such as how a line or curve looks. The filters in the later layers of a CNN learn higher-level features, such as eyes and fingers, and the last few layers learn to identify the exact category. A pre-trained model is an algorithm that is trained on a similar dataset. Most of the popular algorithms are pre-trained on the popular `ImageNet` dataset to identify 1,000 different categories. Such a pre-trained model will have filter weights tuned to identify various patterns. So, let's understand how can we take advantage of these pre-trained weights. We will look into an algorithm called **VGG16**, which was one of the earliest algorithms to find success in ImageNet competitions. Though there are more modern algorithms, this algorithm is still popular as it is simple to understand and use for transfer learning. Let's take a look at the architecture of the VGG16 model, and then try to understand the architecture and how we can use it to train our image classifier:


Convolution2D
Convolution2D
MaxPooling2D
Convolution block 1 with 64 filters
Convolution2D
Convolution2D
MaxPooling2D
Convolution block 2 with 128 filters
Convolution2D
Convolution2D
MaxPooling2D
Convolution block 3 with 256 filters
Convolution2D
Convolution2D
MaxPooling2D
Convolution block 4 with 512 filters
Convolution2D
Convolution2D
MaxPooling2D
Convolution block 5 with 512 filters
Flatten
Linear
Dropout
Fully connected layer. We will learn the parameters or weights only for these layers keeping the weights of the rest of the layers unchanged.
Linear
Softmax

Architecture of the VGG16 model

The VGG16 architecture contains five VGG blocks. A block is a set of convolution layers, a nonlinear activation function, and a max-pooling function. All the algorithm parameters are tuned to achieve state-of-the-art results for classifying 1,000 categories. The algorithm takes input data in the form of batches, which are normalized by the mean and standard deviation of the `ImageNet` dataset. In transfer learning, we try to capture what the algorithm learns by freezing the learned parameters of most of the layers of the architecture. It is often a common practice to fine-tune only the last layers of the network. In this example, let's train only the last few linear layers and leave the convolutional layers intact, as the features learned by the convolutional features are mostly used for all kinds of image problems where the images share similar properties. Let's train a VGG16 model using transfer learning to classify dogs and cats. Let's walk through the different steps required to implement this.

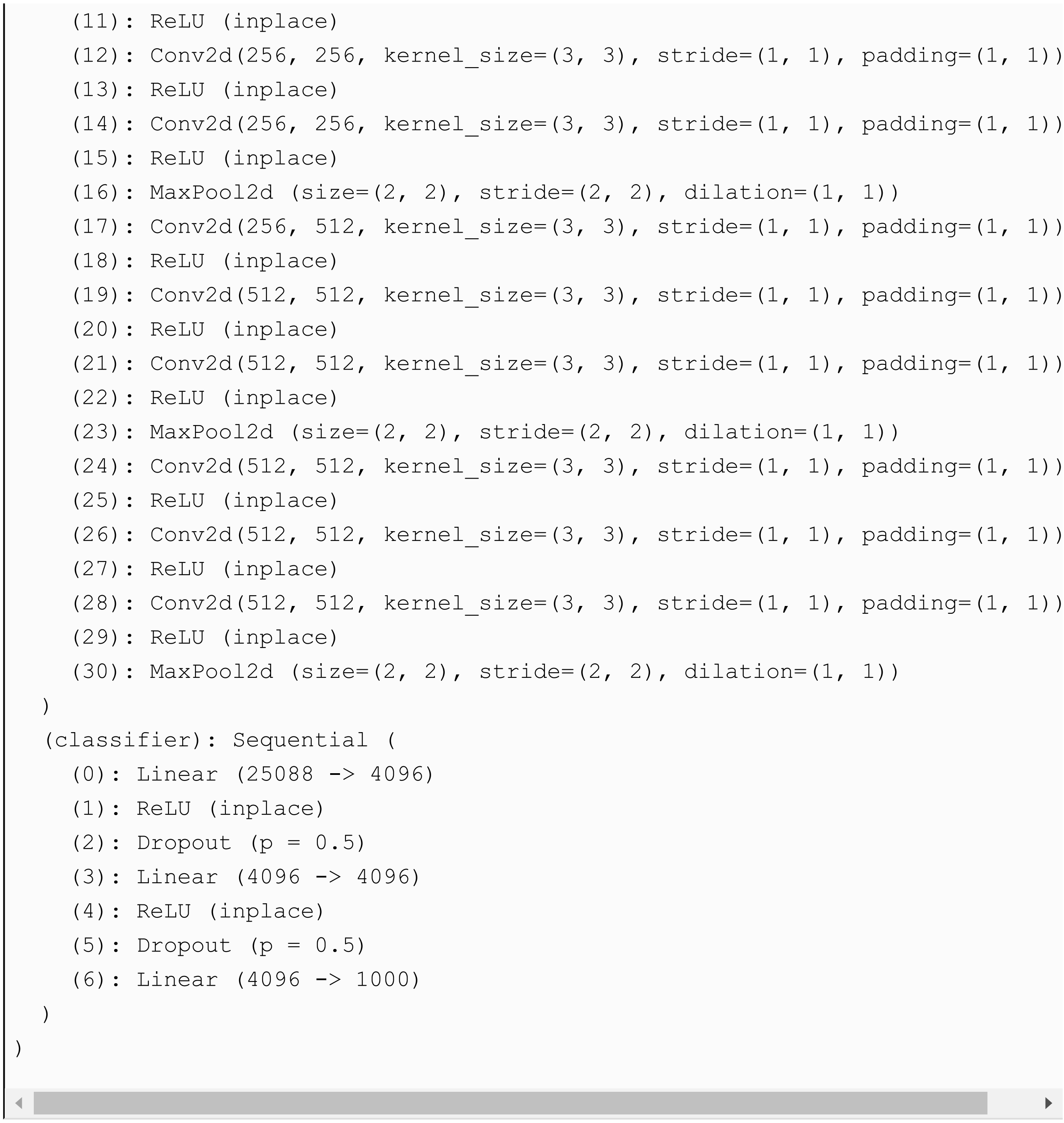
Creating and exploring a VGG16 model

PyTorch provides a set of trained models in its `torchvision` library. Most of them accept an argument called `pretrained` when `True`, which downloads the weights tuned for the **ImageNet** classification problem. Let's look at the code snippet that creates a VGG16 model:

```
from torchvision import models
vgg = models.vgg16(pretrained=True)
```

Now we have our VGG16 model with all the pre-trained weights ready to be used. When the code is run for the first time, it could take several minutes, depending on your internet speed. The size of the weights could be around 500 MB. We can take a quick look at the VGG16 model by printing it. Understanding how these networks are implemented turns out to be very useful when we use modern architectures. Let's take a look at the model:

```
VGG (
  (features): Sequential (
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU (inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU (inplace)
    (4): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU (inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU (inplace)
    (9): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```



```
    (11): ReLU (inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU (inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU (inplace)
    (16): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU (inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU (inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU (inplace)
    (23): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU (inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU (inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU (inplace)
    (30): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
  )
  (classifier): Sequential (
    (0): Linear (25088 -> 4096)
    (1): ReLU (inplace)
    (2): Dropout (p = 0.5)
    (3): Linear (4096 -> 4096)
    (4): ReLU (inplace)
    (5): Dropout (p = 0.5)
    (6): Linear (4096 -> 1000)
  )
)
```

The model summary contains two sequential model `features` and `classifiers`. The `features sequential` model has the layers that we are going to freeze.

Freezing the layers

Let's freeze all the layers of the `features` model, which contains the convolutional block. Freezing the weights in the layers will prevent the weights of these convolutional blocks. As the weights of the model are trained to recognize a lot of important features, our algorithm would be able to do the same from the very first iteration. The ability to use models weights, which were initially trained for a different use case, is called **transfer learning**. Now let's look at how we can freeze the weights, or parameters, of layers:

```
for param in vgg.features.parameters(): param.requires_grad = False
```

This code prevents the optimizer from updating the weights.

Fine-tuning VGG16

The VGG16 model is trained to classify 1,000 categories, but not trained to classify dogs and cats. So, we need to change the output features of the last layer to `2` from `1000`. The following code snippet does it:

```
vgg.classifier[6].out_features = 2
```

The `vgg.classifier` gives access to all the layers inside the sequential model, and the sixth element will contain the last layer. When we train the VGG16 model, we only need the classifier parameters to be trained. So, we pass only the `classifier.parameters` to the optimizer as follows:

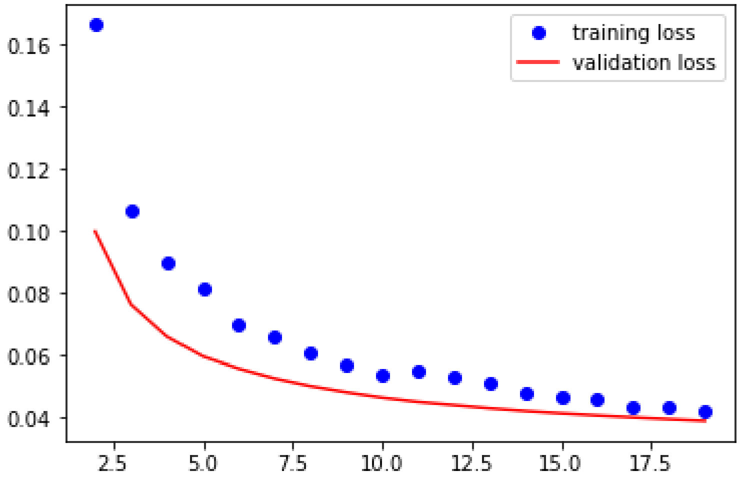
```
optimizer =
  optim.SGD(vgg.classifier.parameters(),lr=0.0001,momentum=0.5)
```

Training the VGG16 model

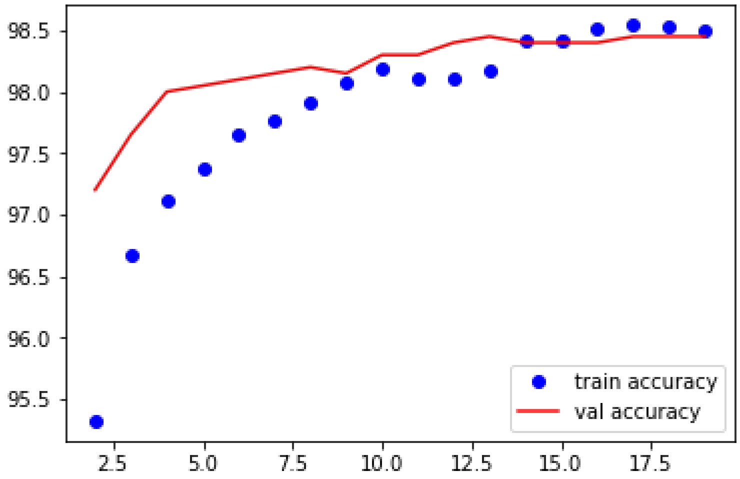
We have created the model and optimizer. Since we are using the `Dogs vs. Cats` dataset, we can use the same data loaders and the `train` function to train our model. Remember, when we train the model, only the parameters inside the classifier change. The following code snippet will train the model for `20` epochs, reaching a validation accuracy of 98.45%:

```
train_losses , train_accuracy = [],[]
val_losses , val_accuracy = [],[]
for epoch in range(1,20):
    epoch_loss, epoch_accuracy = fit(epoch,vgg,train_data_loader,phase='trair
    val_epoch_loss , val_epoch_accuracy = fit(epoch,vgg,valid_data_loader,pha
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    val_losses.append(val_epoch_loss)
    val_accuracy.append(val_epoch_accuracy)
```

Let's visualize the training and validation loss:



Let's visualize the training and validation accuracy:



We can apply a couple of tricks, such as data augmentation and playing with different values of the dropout to improve the model's generalization capabilities. The following code snippet changes the dropout, value in the classifier module of VGG to `0.2` from `0.5` and trains the model:

```
for layer in vgg.classifier.children():
    if(type(layer) == nn.Dropout):
        layer.p = 0.2

#Training
train_losses , train_accuracy = [],[]
val_losses , val_accuracy = [],[]
for epoch in range(1,3):
    epoch_loss, epoch_accuracy = fit(epoch,vgg,train_data_loader,phase='trai
    val_epoch_loss , val_epoch_accuracy = fit(epoch,vgg,valid_data_loader,pha
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    val_losses.append(val_epoch_loss)
    val_accuracy.append(val_epoch_accuracy)
```

Training this for a few epochs gave me a slight improvement; you can try playing with different dropout values. Another important trick to improve model generalization is to add more data or do data augmentation. We will do data augmentation, by randomly flipping the image horizontally or rotating the image by a small angle. The `torchvision` transforms provide different functionalities for doing data augmentation and they do it dynamically, changing for every epoch. We implement data augmentation using the following code:

```
train_transform =transforms.Compose([transforms.Resize((224,224)),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.RandomRotation(0.2),
                                     transforms.ToTensor(),
                                     transforms.Normalize([0.485, 0.456, 0.4
                                      ])

train = ImageFolder('dogsandcats/train/',train_transform)
valid = ImageFolder('dogsandcats/valid/',simple_transform)

#Training

train_losses , train_accuracy = [],[]
val_losses , val_accuracy = [],[]
for epoch in range(1,3):
    epoch_loss, epoch_accuracy = fit(epoch,vgg,train_data_loader,phase='trai
    val_epoch_loss , val_epoch_accuracy = fit(epoch,vgg,valid_data_loader,pha
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    val_losses.append(val_epoch_loss)
    val_accuracy.append(val_epoch_accuracy)
```

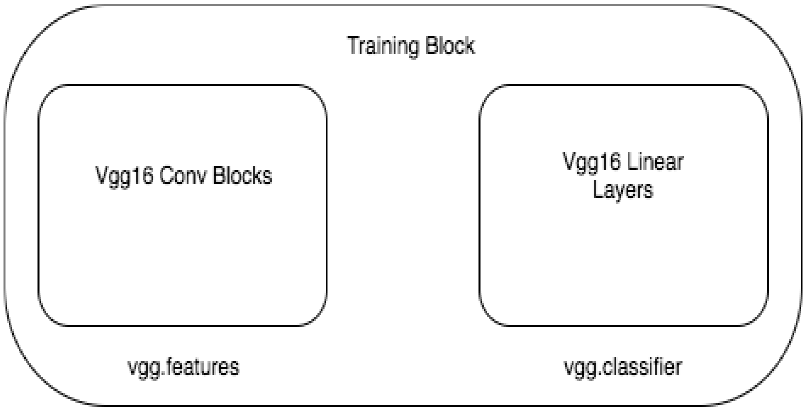
The output of the preceding code is generated as follows:

```
#Results

training loss is 0.041 and training accuracy is 22657/23000 98.51 validation
```

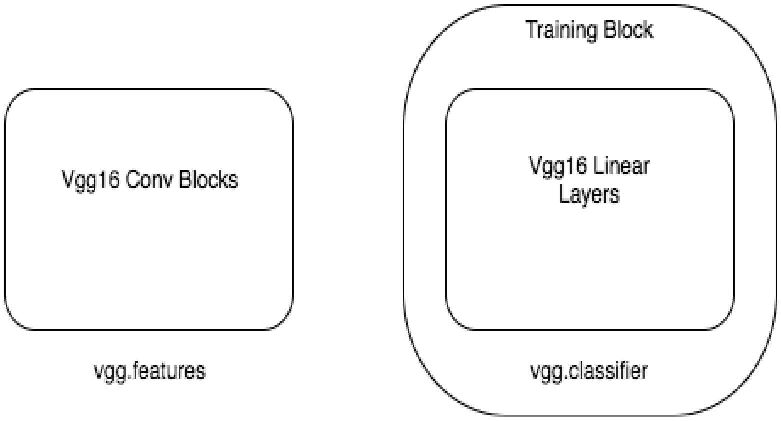
Training the model with augmented data improved the model accuracy by 0.1% by running just two epochs; we can run it for a few more epochs to improve further. If you have been training these models while reading the book, you will have realized that training each epoch could take more than a couple of minutes, depending on the GPU you are running. Let's look at a technique where we can train each epoch in a few seconds.

Calculating pre-convoluted features

When we freeze the convolution layers and the train model, the input to the fully connected layers, or dense layers, (`vgg.classifier`) is always the same. To understand better, let's treat the convolution block, in our example the `vgg.features` block, as a function that has learned weights and does not change during training. So, calculating the convolution features and storing them will help us to improve the training speed. The time to train the model decreases, as we calculate these features only once instead of calculating for each epoch. Let's visually understand and implement the same:


Training Block
Vgg16 Conv Blocks
Vgg16 Linear Layers
vgg.features
vgg.classifier

During training we run both the features block and classifier block though the output of the vgg.features remain constant as the weights are freezed


Training Block
Vgg16 Conv Blocks
Vgg16 Linear Layers
vgg.features
vgg.classifier

We calculate the output of Conv blocks once and use that for training the Linear layers.

The first box depicts how training is done in general, which could be slow, as we calculate the convolutional features for every epoch, though the values do not change. In the bottom box, we calculate the convolutional features once and train only the linear layers. To calculate the pre-convoluted features, we will pass all the training data through the convolution blocks and store them. To perform this, we need to select the convolution blocks of the VGG model. Fortunately, the PyTorch implementation of VGG16 has two sequential models, so just picking the first sequential model's features is enough. The following code does that:

```
vgg = models.vgg16(pretrained=True)
vgg = vgg.cuda()
features = vgg.features

train_data_loader = torch.utils.data.DataLoader(train,batch_size=32,num_worke
valid_data_loader = torch.utils.data.DataLoader(valid,batch_size=32,num_worke
def preconvfeat(dataset,model):
    conv_features = []
    labels_list = []
    for data in dataset:
        inputs,labels = data
        if is_cuda:
            inputs , labels = inputs.cuda(),labels.cuda()
        inputs , labels = Variable(inputs),Variable(labels)
        output = model(inputs)
        conv_features.extend(output.data.cpu().numpy())
        labels_list.extend(labels.data.cpu().numpy())
    conv_features = np.concatenate([[feat] for feat in conv_features])

    return (conv_features,labels_list)

conv_feat_train,labels_train = preconvfeat(train_data_loader,features)
conv_feat_val,labels_val = preconvfeat(valid_data_loader,features)
```

In the previous code, the `preconvfeat` method takes in the dataset and `vgg` model and returns the convoluted features along with the labels associated with it. The rest of the code is similar to what we have used in the other examples for creating data loaders and datasets.

Once we have the convolutional features for the `train` and `validation` sets, let's create a PyTorch dataset and `DataLoader` classes, which will ease up our training process. The following code creates the `Dataset` and `DataLoader` for our convolution features:

```
class My_dataset(Dataset):
    def __init__(self,feat,labels):
        self.conv_feat = feat
        self.labels = labels

    def __len__(self):
        return len(self.conv_feat)

    def __getitem__(self,idx):
        return self.conv_feat[idx],self.labels[idx]

train_feat_dataset = My_dataset(conv_feat_train,labels_train)
val_feat_dataset = My_dataset(conv_feat_val,labels_val)
```

```
train_feat_loader =
  DataLoader(train_feat_dataset,batch_size=64,shuffle=True)
val_feat_loader =
  DataLoader(val_feat_dataset,batch_size=64,shuffle=True)
```

As we have our new data loaders that generate batches of convoluted features along with the labels, we can use the same `train` function that we have been using in the other examples. Now we will use `vgg.classifier` as the model for creating the `optimizer` and `fit` methods. The following code trains the classifier module to identify dogs and cats. On a Titan X GPU, each epoch takes less than five seconds, which would otherwise take a few minutes:

```
train_losses , train_accuracy = [],[]
val_losses , val_accuracy = [],[]
for epoch in range(1,20):
    epoch_loss, epoch_accuracy = fit_numpy(epoch,vgg.classifier,train_feat_lo
    val_epoch_loss , val_epoch_accuracy = fit_numpy(epoch,vgg.classifier,val_
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    val_losses.append(val_epoch_loss)
    val_accuracy.append(val_epoch_accuracy)
```

Understanding what a CNN model learns

Deep learning models are often said to be not interpretable. But there are different techniques that are being explored to interpret what happens inside these models. For images, the features learned by convents are interpretable. We will explore two popular techniques to understand convents.

Visualizing outputs from intermediate layers

Visualizing the outputs from intermediate layers will help us in understanding how the input image is being transformed across different layers. Often, the output from each layer is called an **activation**. To do this, we should extract output from intermediate layers, which can be done in different ways. PyTorch provides a method called `register_forward_hook`, which allows us to pass a function which can extract outputs of a particular layer.

By default, PyTorch models only store the output of the last layer, to use memory optimally. So, before we inspect what the activations from the intermediate layers look like, let's understand how to extract outputs from the model. Let's look at the following code snippet, which extracts, and we will walk through it to understand what happens:

```
vgg = models.vgg16(pretrained=True).cuda()

class LayerActivations():
    features=None

    def __init__(self,model,layer_num):
        self.hook = model[layer_num].register_forward_hook(self.hook_fn)

    def hook_fn(self,module,input,output):
        self.features = output.cpu()

    def remove(self):
        self.hook.remove()
```

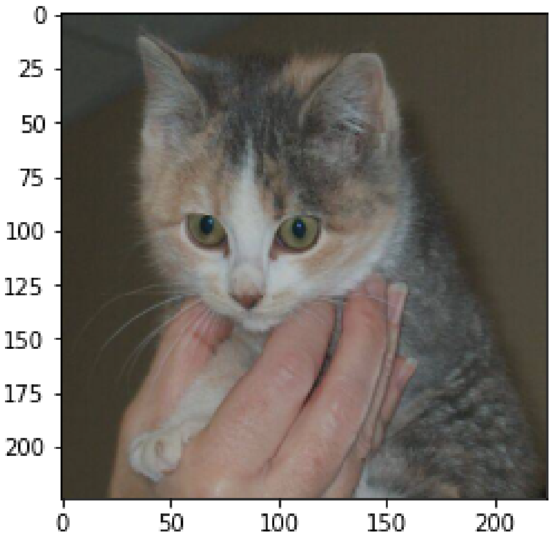
```
conv_out = LayerActivations(vgg.features,0)

o = vgg(Variable(img.cuda()))

conv_out.remove()

act = conv_out.features
```

We start with the creation of a pre-trained VGG model, from which we extract the outputs of a particular layer. The `LayerActivations` class instructs PyTorch to store the output of a layer to the `features` variable. Let's walk through each function inside the `LayerActivations` class.

The `_init_` function takes a model and the layer number for which the outputs need to be extracted as arguments. We call the `register_forward_hook` method on the layer and pass in a function. PyTorch, when doing a forward pass—that is, when the images are passed through the layers—calls the function that is passed to the `register_forward_hook` method. This method returns a handle, which can be used to deregister the function that is passed to the `register_forward_hook` method.

The `register_forward_hook` method passes three values to the function that we pass to it. The `module` parameter allows us access to the layer itself. The second parameter is `input`, which refers to data that is flowing through the layer. The third parameter is `output`, which allows access to the transformed inputs, or activation, of the layer. We store the output to the features variable in the `LayerActivations` class.

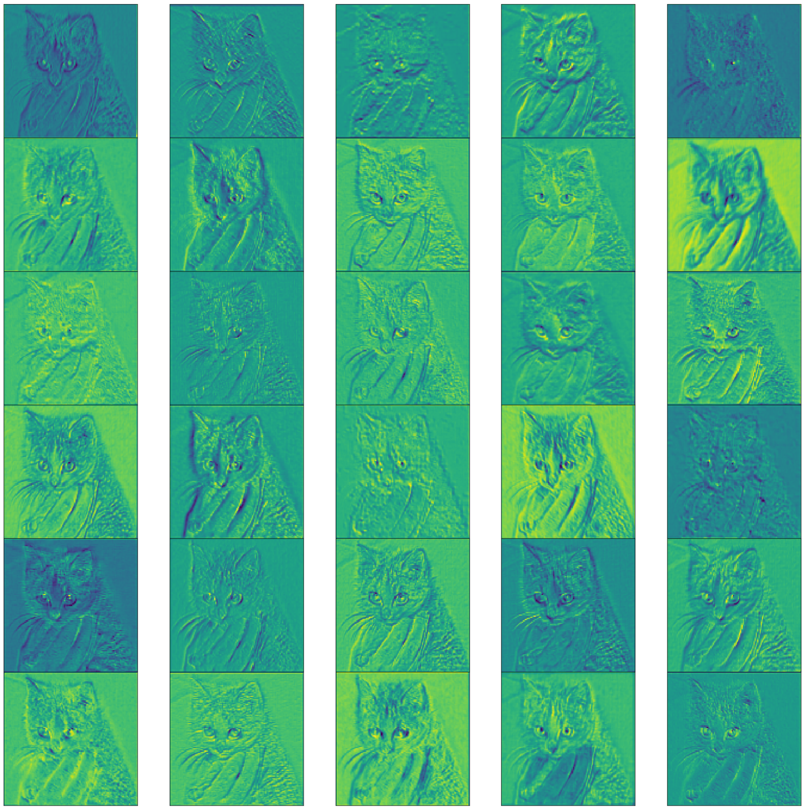
The third function takes the `hook` from the `_init_` function and deregisters the function. Now we can pass the model and the layer number for which we are looking for activations. Let's look at the activations created for the following image for different layers:


0
25
50
75
100
125
150
175
200
0
50
100
150
200

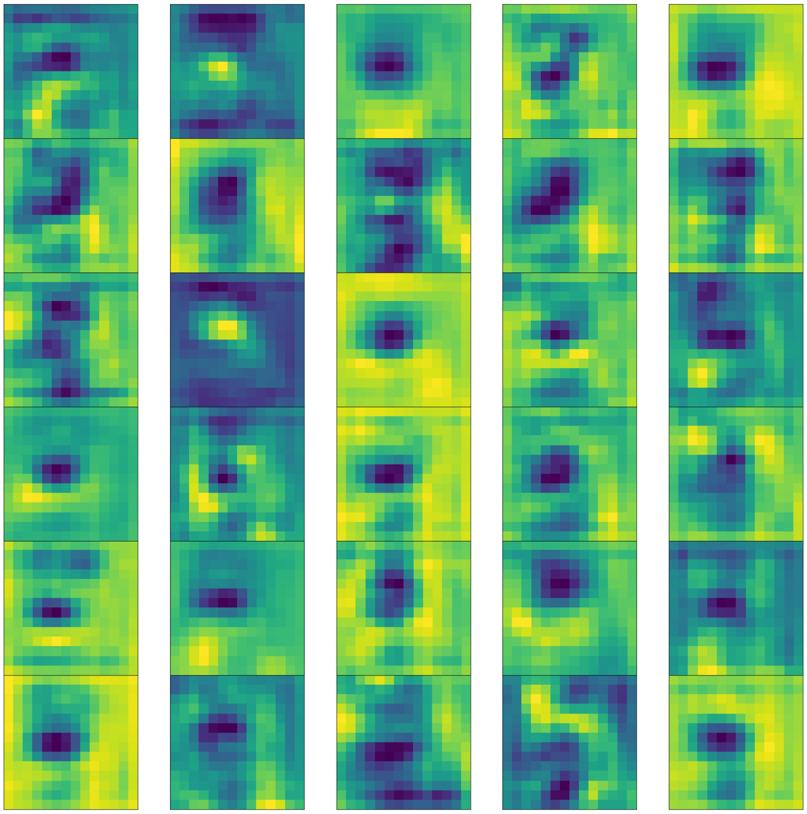
Let's visualize some of the activations created by the first convolution layer and the code used for it:

```
fig = plt.figure(figsize=(20,50))
fig.subplots_adjust(left=0,right=1,bottom=0,top=0.8,hspace=0,
  wspace=0.2)
for i in range(30):
    ax = fig.add_subplot(12,5,i+1,xticks=[],yticks=[])
    ax.imshow(act[0][i])
```

Let's visualize some of the activations created by the fifth convolution layer:



Let's look at the last CNN layer:



From looking at what different layers generate, we can see that the early layers detect lines and edges, and the last layers tend to learn higher-level features and are less interpretable. Before we move on to visualizing weights, let's see how the features maps or activations represents itself after the ReLU layer. So, let's visualize the outputs of the second layer.

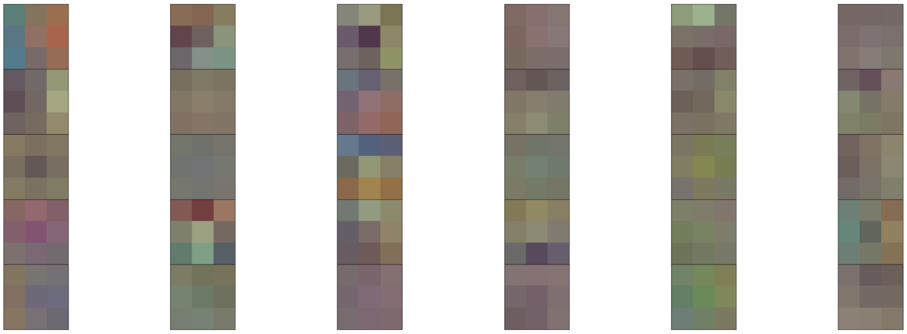
If you take a quick look at the fifth image in the second row of the preceding image, it looks like the filter is detecting the eyes in the image. When the models do not perform, these tricks to visualize can help us understand why the model may not be working.

Visualizing weights of the CNN layer

Getting model weights for a particular layer is straightforward. All the model weights can be accessed through the `state_dict` function. The `state_dict` function returns a dictionary, with keys as its layers and weights as its values. The following code demonstrates how to pull weights for a particular layer and visualize them:

```
vgg.state_dict().keys()
cnn_weights = vgg.state_dict()['features.0.weight'].cpu()
```

The preceding code provides us with the following output:



Each box represents weights of a filter that is of size *3 x 3*. Each filter is trained to identify certain patterns in the images.

Summary

In this chapter, we learned how to build an image classifier using convents, and also how to use a pre-trained model. We covered tricks on how to speed up the process of training by using these pre-convoluted features. Also, we understood different techniques that can be used to understand what goes on inside a CNN.

In the next chapter, we will learn how to handle sequential data using recurrent neural networks.

Deep Learning with Sequence Data and Text

In the last chapter, we covered how to handle spatial data using **Convolution Neural Networks** (**CNNs**) and also built image classifiers. In this chapter, we will cover the following topics:

- Different representations of text data that are useful for building deep learning models
- Understanding **recurrent neural networks** (**RNNs**) and different implementations of RNNs, such as **Long Short-Term Memory** (**LSTM**) and **Gated Recurrent Unit** (**GRU**), which power most of the deep learning models for text and sequential data
- Using one-dimensional convolutions for sequential data

Some of the applications that can be built using RNNs are:

- **Document classifiers**: Identifying the sentiment of a tweet or review, classifying news articles
- **Sequence-to-sequence learning**: For tasks such as language translations, converting English to French

- **Time-series forecasting**: Predicting the sales of a store given details about previous days' store details

Working with text data

Text is one of the commonly used sequential data types. Text data can be seen as either a sequence of characters or a sequence of words. It is common to see text as a sequence of words for most problems. Deep learning sequential models such as RNN and its variants are able to learn important patterns from text data that can solve problems in areas such as:

- Natural language understanding
- Document classification
- Sentiment classification

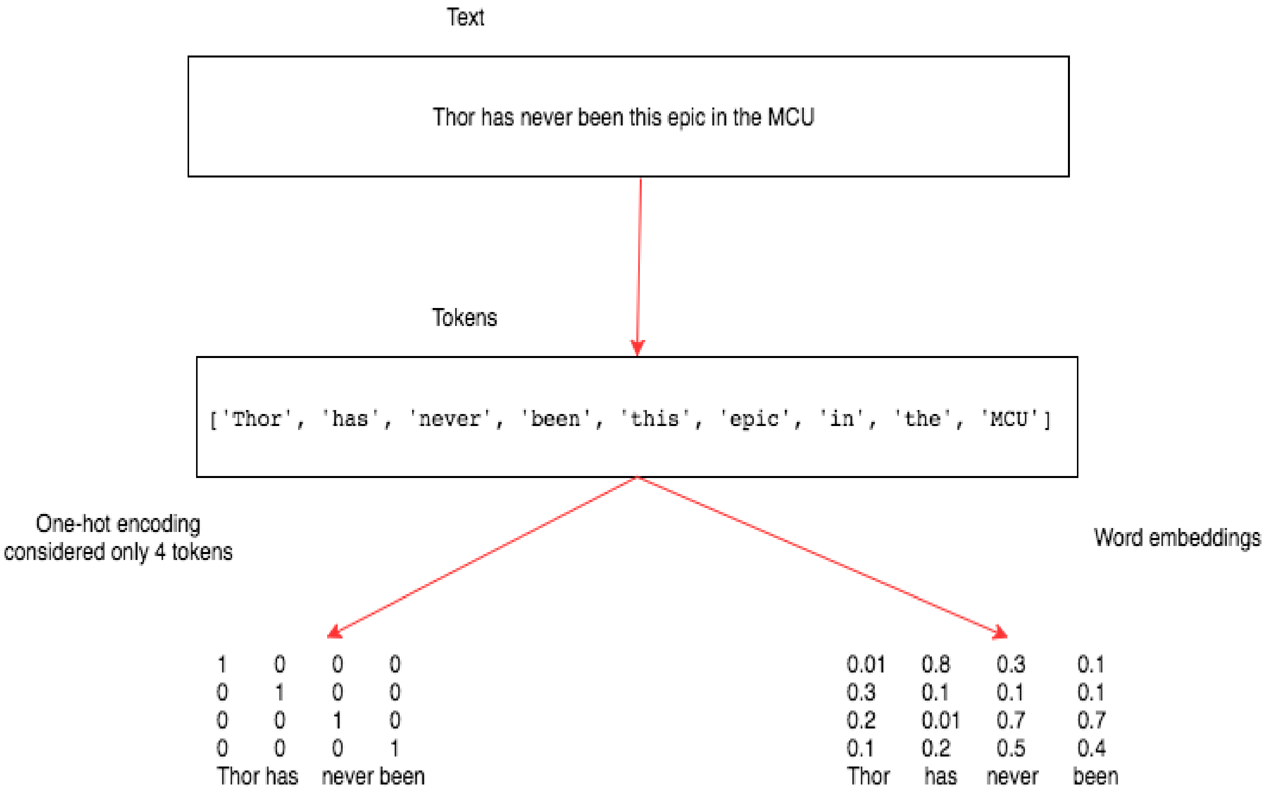
These sequential models also act as important building blocks for various systems, such as **question and answering** (**QA**) systems.

Though these models are highly useful in building these applications, they do not have an understanding of human language, due to its inherent complexities. These sequential models are able to successfully find useful patterns that are then used for performing different tasks. Applying deep learning to text is a fast-growing field, and a lot of new techniques arrive every month. We will cover the fundamental components that power most of the modern-day deep learning applications.

Deep learning models, like any other machine learning model, do not understand text, so we need to convert text into numerical representation. The process of converting text into numerical representation is called **vectorization** and can be done in different ways, as outlined here:

- Convert text into words and represent each word as a vector
- Convert text into characters and represent each character as a vector
- Create *n*-gram of words and represent them as vectors

Text data can be broken down into one of these representations. Each smaller unit of text is called a **token**, and the process of breaking text into tokens is called **tokenization**. There are a lot of powerful libraries available in Python that can help us in tokenization. Once we convert the text data into tokens, we then need to map each token to a vector. One-hot encoding and word embedding are the two most popular approaches for mapping tokens to vectors. The following diagram summarizes the steps for converting text into their vector representations:


Text
Thor has never been this epic in the MCU
Tokens
['Thor', 'has', 'never', 'been', 'this', 'epic', 'in', 'the', 'MCU']
One-hot encoding
considered only 4 tokens
Word embeddings
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
Thor has never been
0.01 0.8 0.3 0.1
0.3 0.1 0.1 0.1
0.2 0.01 0.7 0.7
0.1 0.2 0.5 0.4
Thor has never been

Let's look in more detail at tokenization, *n*-gram representation, and vectorization.

Tokenization

Given a sentence, splitting it into either characters or words is called **tokenization**. There are libraries, such as spaCy, that offer complex solutions to tokenization. Let's use simple Python functions such as `split` and `list` to convert the text into tokens.

To demonstrate how tokenization works on characters and words, let's consider a small review of the movie *Thor: Ragnarok*. We will work with the following text:

*The action scenes were top notch in this movie. Thor has never been this epic in the MCU. He does some pretty epic sh***t in this movie and he is definitely not under-powered anymore. Thor in unleashed in this, I love that.*

Converting text into characters

The Python `list` function takes a string and converts it into a list of individual characters. This does the job of converting the text into characters. The following code block shows the code used and the results:

```
thor_review = "the action scenes were top notch in this movie. Thor has never

print(list(thor_review))
```

The result is as follows:

```
#Results
['t', 'h', 'e', ' ', 'a', 'c', 't', 'i', 'o', 'n', ' ', 's', 'c', 'e', 'n', '
```

This result shows how our simple Python function has converted text into tokens.

Converting text into words

We will use the `split` function available in the Python string object to break the text into words. The `split` function takes an argument, based on which it splits the text into tokens. For our example, we will use spaces as the delimiters. The following code block demonstrates how we can convert text into words using the Python `split` function:

```
print(Thor_review.split())

#Results

['the', 'action', 'scenes', 'were', 'top', 'notch', 'in', 'this', 'movie.', '
```

In the preceding code, we did not use any separator; by default, the `split` function splits on white spaces.

N-gram representation

We have seen how text can be represented as characters and words. Sometimes it is useful to look at two, three, or more words together. *N*-grams are groups of words extracted from given text. In an *n*-gram, *n* represents the number of words that can be used together. Let's look at an example of what a bigram (*n=2*) looks like. We used the Python `nltk` package to generate a bigram for `thor_review`. The following code block shows the result of the bigram and the code used to generate it:

```
from nltk import ngrams

print(list(ngrams(thor_review.split(),2)))

#Results
[('the', 'action'), ('action', 'scenes'), ('scenes', 'were'), ('were', 'top')
```

The `ngrams` function accepts a sequence of words as its first argument and the number of words to be grouped as the second argument. The following code block shows how a trigram representation would look, and the code used for it:

```
print(list(ngrams(thor_review.split(),3)))

#Results

[('the', 'action', 'scenes'), ('action', 'scenes', 'were'), ('scenes', 'were'
```

The only thing that changed in the preceding code is the *n*-value, the second argument to the function.

Many supervised machine learning models, for example, Naive Bayes, use *n*-grams to improve their feature space. *n*-grams are also used for spelling correction and text-summarization tasks.

One challenge with *n*-gram representation is that it loses the sequential nature of text. It is often used with shallow machine learning models. This technique is rarely used in deep learning, as architectures such as RNN and Conv1D learn these representations automatically.

Vectorization

There are two popular approaches to mapping the generated tokens to vectors of numbers, called **one-hot encoding** and **word embedding**. Let's understand how tokens can be converted to these vector representations by writing a simple Python program. We will also discuss the various pros and cons of each method.

One-hot encoding

In one-hot encoding, each token is represented by a vector of length N, where *N* is the size of the vocabulary. The vocabulary is the total number of unique words in the document. Let's take a simple sentence and observe how each token would be represented as one-hot encoded vectors. The following is the sentence and its associated token representation:

An apple a day keeps doctor away said the doctor.

One-hot encoding for the preceding sentence can be represented into a tabular format as follows:

An	100000000
apple	010000000
a	001000000
day	000100000
keeps	000010000
doctor	000001000
away	000000100
said	000000010
the	000000001

This table describes the tokens and their one-hot encoded representation. The vector length is 9, as there are nine unique words in the sentence. A lot of machine learning libraries have eased the process of creating one-hot encoding variables. We will write our own implementation to make it easier to understand, and we can use the same implementation to build other features required for later examples. The following code contains a `Dictionary` class, which contains functionality to create a dictionary of unique words along with a function to return a one-hot encoded vector for a particular word. Let's take a look at the code and then walk through each functionality:

```
class Dictionary(object):
    def __init__(self):
        self.word2idx = {}
        self.idx2word = []
        self.length = 0

    def add_word(self,word):
        if word not in self.idx2word:
            self.idx2word.append(word)
            self.word2idx[word] = self.length + 1
            self.length += 1
        return self.word2idx[word]

    def __len__(self):
        return len(self.idx2word)

    def onehot_encoded(self,word):
        vec = np.zeros(self.length)
        vec[self.word2idx[word]] = 1
        return vec
```

The preceding code provides three important functionalities:

- The initialization function, `__init__`, creates a `word2idx` dictionary, which will store all unique words along with the index. The `idx2word` list stores all the unique words, and the `length` variable contains the total number of unique words in our documents.
- The `add_word` function takes a word and adds it to `word2idx` and `idx2word`, and increases the length of the vocabulary, provided the word is unique.
- The `onehot_encoded` function takes a word and returns a vector of length N with zeros throughout, except at the index of the word. If the index of the passed word is two, then the value of the vector at index two will be one, and all the remaining values will be zeros.

As we have defined our `Dictionary` class, let's use it on our `thor_review` data. The following code demonstrates how the `word2idx` is built and how we can call our `onehot_encoded` function:

```
dic = Dictionary()

for tok in thor_review.split():
    dic.add_word(tok)

print(dic.word2idx)
```

The output of the preceding code is as follows:

```
# Results of word2idx

{'the': 1, 'action': 2, 'scenes': 3, 'were': 4, 'top': 5, 'notch': 6, 'in': 7
```

One-hot encoding for the word `were` is as follows:

```
# One-hot representation of the word 'were'
dic.onehot_encoded('were')
array([ 0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

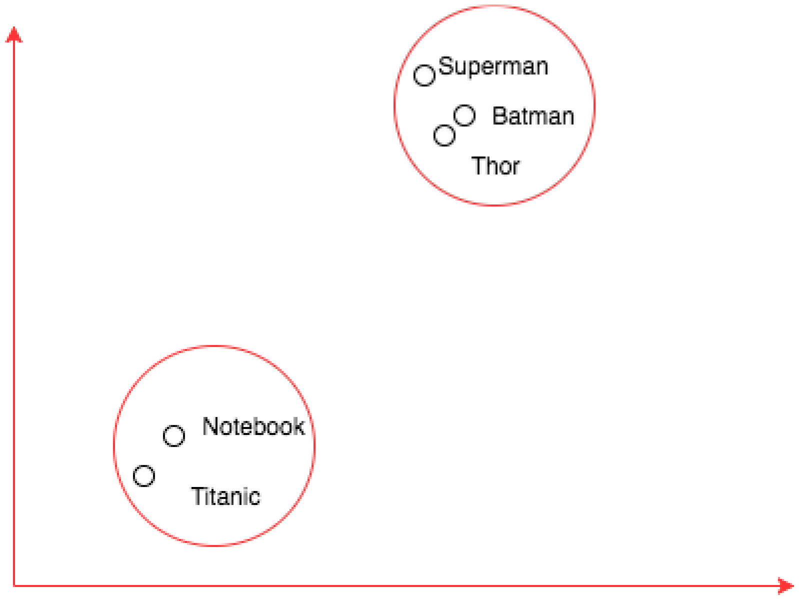
One of the challenges with one-hot representation is that the data is too sparse, and the size of the vector quickly grows as the number of unique words in the vocabulary increases, which is considered to be a limitation, and hence it is rarely used with deep learning.

Word embedding

Word embedding is a very popular way of representing text data in problems that are solved by deep learning algorithms. Word embedding provides a dense representation of a word filled with floating numbers. The vector dimension varies according to the vocabulary size. It is common to use a word embedding of dimension size 50, 100, 256, 300, and sometimes 1,000. The dimension size is a hyper-parameter that we need to play with during the training phase.

If we are trying to represent a vocabulary of size 20,000 in one-hot representation then we will end up with 20,000 x 20,000 numbers, most of which will be zero. The same vocabulary can be represented in word embedding as 20,000 x dimension size, where the dimension size could be 10, 50, 300, and so on.

One way to create word embeddings is to start with dense vectors for each token containing random numbers, and then train a model such as a document classifier or sentiment classifier. The floating point numbers, which represent the tokens, will get adjusted in a way such that semantically closer words will have similar representation. To understand it, let's look at the following figure, where we plotted the word embedding vectors on a two-dimensional plot of five movies:



The preceding image shows how the dense vectors are tuned in order to have smaller distances for words that are semantically similar. Since movie titles such as **Superman**, **Thor**, and **Batman** are action movies based on comics, the embedding for such words is closer, whereas the embedding for the movie **Titanic** is far from the action movies and closer to the movie title **Notebook**, since they are romantic movies.

Learning word embedding may not be feasible when you have too little data, and in such cases we can use word embeddings that are trained by some other machine learning algorithm. An embedding generated from another task is called a **pretrained** word embedding. We will learn how to build our own word embeddings and use pretrained word embeddings.

Training word embedding by building a sentiment classifier

In the last section, we briefly learned about word embedding without implementing it. In this section, we will download a dataset called `IMDB`, which contains reviews, and build a sentiment classifier which calculates whether a review's sentiment is positive, negative, or unknown. In the process of building, we will also train word embedding for the words present in the `IMDB` dataset. We will use a library called `torchtext` that makes a lot of processes such as downloading, text vectorization, and batching much easier. Training a sentiment classifier will involve the following steps:

1. Downloading IMDB data and performing text tokenization
2. Building a vocabulary
3. Generating batches of vectors
4. Creating a network model with embeddings
5. Training the model

Downloading IMDB data and performing text tokenization

For applications related to computer vision, we used the `torchvision` library, which provides us with a lot of utility functions, helping to building computer vision applications. In the same way, there is a library called `torchtext`, part of PyTorch, which is built to work with PyTorch and eases a lot of activities related to **natural language processing** (**NLP**) by providing different data loaders and abstractions for text. At the time of writing, `torchtext` does not come with PyTorch installation and requires a separate installation. You can run the following code in the command line of your machine to get `torchtext` installed:

```
pip install torchtext
```

Once it is installed, we will be able to use it. Torchtext provides two important modules called `torchtext.data` and `torchtext.datasets`.

> *We can download the `IMDB Movies` dataset from the following link:*
>
> *https://www.kaggle.com/orgesleka/imdbmovies*

torchtext.data

The `torchtext.data` instance defines a class called `Field`, which helps us to define how the data has to be read and tokenized. Let's look at the following example, which we will use for preparing our `IMDB` dataset:

```
from torchtext import data
TEXT = data.Field(lower=True, batch_first=True,fix_length=20)
LABEL = data.Field(sequential=False)
```

In the preceding code, we define two `Field` objects, one for actual text and another for the label data. For actual text, we expect `torchtext` to lowercase all the text, tokenize the text, and trim it to a maximum length of `20`. If we are building the application for a production environment, we may fix the length to a much larger number. But, for the toy example it works well. The `Field` constructor also accepts another argument called **tokenize**, which by default uses the `str.split` function. We can also specify spaCy as the argument, or any other tokenizer. For our example we will stick with `str.split`.

torchtext.datasets

The `torchtext.datasets` instance provide wrappers for using different datasets like IMDB, TREC (question classification), language modeling (WikiText-2), and a few other datasets. We will use `torch.datasets` to download the `IMDB` dataset and split it into `train` and `test` datasets. The following code does that, and when you run it for the first time it could take several minutes, depending on your broadband connection, as it downloads the `IMDB` datasets from the internet:

```
train, test = datasets.IMDB.splits(TEXT, LABEL)
```

The previous dataset's `IMDB` class abstracts away all the complexity involved in downloading, tokenizing, and splitting the database into `train` and `test` datasets. `train.fields` contains a dictionary where `TEXT` is the key and the value `LABEL`. Let's look at `train.fields` and each element of `train` contains:

```
print('train.fields', train.fields)

#Results
train.fields {'text': <torchtext.data.field.Field object at 0x1129db160>, 'la

print(vars(train[0]))

#Results

vars(train[0]) {'text': ['for', 'a', 'movie', 'that', 'gets', 'no', 'respect'
```

We can see from these results that a single element contains a field, `text`, along with all the tokens representing the `text`, and a `label` field that contains the label of the text. Now we have the `IMDB` dataset ready for batching.

Building vocabulary

When we created one-hot encoding for `thor_review`, we created a `word2idx` dictionary, which is referred to as the vocabulary since it contains all the details of the unique words in the documents. The `torchtext` instance makes that easier for us. Once the data is loaded, we can call `build_vocab` and pass the necessary arguments that will take care of building the vocabulary for the data. The following code shows how the vocabulary is built:

```
TEXT.build_vocab(train, vectors=GloVe(name='6B', dim=300),max_size=10000,min_
LABEL.build_vocab(train)
```

In the preceding code, we pass in the `train` object on which we need to build the vocabulary, and we also ask it to initialize vectors with pretrained embeddings of dimensions `300`. The `build_vocab` object just downloads and creates the dimension that will be used later, when we train the sentiment classifier using pretrained weights. The `max_size` instance limits the number of words in the vocabulary, and `min_freq` removes any word which has not occurred more than ten times, where `10` is configurable.

Once the vocabulary is built, we can obtain different values such as frequency, word index, and the vector representation for each word. The following code demonstrates how to access these values:

```
print(TEXT.vocab.freqs)

# A sample result
Counter({"i'm": 4174,
         'not': 28597,
         'tired': 328,
         'to': 133967,
         'say': 4392,
         'this': 69714,
         'is': 104171,
         'one': 22480,
         'of': 144462,
         'the': 322198,
```

The following code demonstrates how to access the results:

```
print(TEXT.vocab.vectors)

#Results displaying the 300 dimension vector for each word.
0.0000 0.0000 0.0000 ... 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 ... 0.0000 0.0000 0.0000
0.0466 0.2132 -0.0074 ... 0.0091 -0.2099 0.0539
      ... ⋱ ...
0.0000 0.0000 0.0000 ... 0.0000 0.0000 0.0000
0.7724 -0.1800 0.2072 ... 0.6736 0.2263 -0.2919
0.0000 0.0000 0.0000 ... 0.0000 0.0000 0.0000
[torch.FloatTensor of size 10002x300]

print(TEXT.vocab.stoi)

# Sample results
defaultdict(<function torchtext.vocab._default_unk_index>,
            {'<unk>': 0,
             '<pad>': 1,
             'the': 2,
             'a': 3,
             'and': 4,
             'of': 5,
             'to': 6,
             'is': 7,
             'in': 8,
             'i': 9,
             'this': 10,
             'that': 11,
             'it': 12,
```

The `stoi` gives access to a dictionary containing words and their indexes.

Generate batches of vectors

Torchtext provides `BucketIterator`, which helps in batching all the text and replacing the words with the index number of the words. The `BucketIterator` instance comes with a lot of useful parameters like `batch_size`, `device` (GPU or CPU), and `shuffle` (whether data has to be shuffled). The following code demonstrates how to create iterators that generate batches for the `train` and `test` datasets:

```
train_iter, test_iter = data.BucketIterator.splits((train, test), batch_size=
#device = -1 represents cpu , if u want gpu leave it to None.
```

The preceding code gives a `BucketIterator` object for `train` and `test` datasets. The following code will show how to create a `batch` and display the results of the `batch`:

```
batch = next(iter(train_iter))
batch.text

#Results
Variable containing:
 5128 427 19 ... 1688 0 542
   58 2 0 ... 2 0 1352
    0 9 14 ... 2676 96 9
       ... ⋱ ...
  129 1181 648 ... 45 0 2
 6484 0 627 ... 381 5 2
  748 0 5052 ... 18 6660 9827
[torch.LongTensor of size 128x20]

batch.label

#Results
Variable containing:
 2
 1
 2
 1
 2
 1
 1
 1
[torch.LongTensor of size 128]
```

From the results in the preceding code block, we can see how the text data is converted into a matrix of size (`batch_size` * `fix_len`), which is (`128x20`) .

Creating a network model with embedding

We discussed word embeddings briefly earlier. In this section, we create word embeddings as part of our network architecture and train the entire model to predict the sentiment of each review. At the end of the training, we will have a sentiment classifier model and also the word embeddings for the `IMDB` datasets. The following code demonstrates how to create a network architecture to predict the sentiment using word embeddings:

```
class EmbNet(nn.Module):
    def __init__(self,emb_size,hidden_size1,hidden_size2=400):
        super().__init__()
        self.embedding = nn.Embedding(emb_size,hidden_size1)
        self.fc = nn.Linear(hidden_size2,3)

    def forward(self,x):
        embeds = self.embedding(x).view(x.size(0),-1)
        out = self.fc(embeds)
        return F.log_softmax(out,dim=-1)
```

In the preceding code, `EmbNet` creates the model for sentiment classification. Inside the `__init__` function, we initialize an object of the `nn.Embedding` class, which takes two arguments, namely, the size of the vocabulary and the dimensions that we wish to create for each word. As we have limited the number of unique words, the vocabulary size will be 10,000 and we can start with a small embedding size of `10`. For running the program quickly, a small embedding size is useful, but when you are trying to build applications for production systems, use embeddings of a large size.

We also have a linear layer that maps the word embeddings to the category (positive, negative, or unknown).

The `forward` function determines how the input data is processed. For a batch size of 32 and sentences of a maximum length of 20 words, we will have inputs of the shape 32 x 20. The first embedding layer acts as a lookup table, replacing each word with the corresponding embedding vector. For an embedding dimension of 10, the output becomes 32 x 20 x 10 as each word is replaced with its corresponding embedding. The `view()` function will flatten the result from the embedding layer. The first argument passed to `view` will keep that dimension intact. In our case, we do not want to combine data from different batches, so we preserve the first dimension and flatten the rest of the values in the tensor. After the `view` function is applied, the tensor shape changes to 32 x 200. A dense layer maps the flattened embeddings to the number of categories. Once the network is defined, we can train the network as usual.

> *Remember that in this network, we lose the sequential nature of the text and we just use them as a bag of words.*

Training the model

Training the model is very similar to what we saw for building image classifiers, so we will be using the same functions. We pass batches of data through the model, calculate the outputs and losses, and then optimize the model weights, which includes the embedding weights. The following code does this:

```
def fit(epoch,model,data_loader,phase='training',volatile=False):
    if phase == 'training':
        model.train()
    if phase == 'validation':
        model.eval()
        volatile=True
    running_loss = 0.0
    running_correct = 0
    for batch_idx , batch in enumerate(data_loader):
        text , target = batch.text , batch.label
        if is_cuda:
            text,target = text.cuda(),target.cuda()

        if phase == 'training':
            optimizer.zero_grad()
        output = model(text)
        loss = F.nll_loss(output,target)

        running_loss += F.nll_loss(output,target,size_average=False).data[0]
        preds = output.data.max(dim=1,keepdim=True)[1]
        running_correct += preds.eq(target.data.view_as(preds)).cpu().sum()
        if phase == 'training':
            loss.backward()
            optimizer.step()

    loss = running_loss/len(data_loader.dataset)
    accuracy = 100. * running_correct/len(data_loader.dataset)

    print(f'{phase} loss is {loss:{5}.{2}} and {phase} accuracy is {running_c
    return loss,accuracy
```

```
train_losses , train_accuracy = [],[]
val_losses , val_accuracy = [],[]

train_iter.repeat = False
test_iter.repeat = False

for epoch in range(1,10):

    epoch_loss, epoch_accuracy = fit(epoch,model,train_iter,phase='training')
    val_epoch_loss , val_epoch_accuracy = fit(epoch,model,test_iter,phase='va
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    val_losses.append(val_epoch_loss)
    val_accuracy.append(val_epoch_accuracy)
```

In the preceding code, we call the `fit` method by passing the `BucketIterator` object that we created for batching the data. The iterator, by default, does not stop generating batches, so we have to set the `repeat` variable of the `BucketIterator` object to `False`. If we don't set the `repeat` variable to `False` then the `fit` function will run indefinitely. Training the model for around 10 epochs gives a validation accuracy of approximately 70%.

Using pretrained word embeddings

Pretrained word embeddings would be useful when we are working in specific domains, such as medicine and manufacturing, where we have lot of data to train the embeddings. When we have little data on which we cannot meaningfully train the embeddings, we can use embeddings, which are trained on different data corpuses such as Wikipedia, Google News and Twitter tweets. A lot of teams have open source word embeddings trained using different approaches. In this section, we will explore how `torchtext` makes it easier to use different word embeddings, and how to use them in our PyTorch models. It is similar to transfer learning, which we use in computer vision applications. Typically, using pretrained embedding would involve the following steps:

- Downloading the embeddings
- Loading the embeddings in the model
- Freezing the embedding layer weights

Let's explore in detail how each step is implemented.

Downloading the embeddings

The `torchtext` library abstracts away a lot of complexity involved in downloading the embeddings and mapping them to the right word. Torchtext provides three classes, namely `GloVe`, `FastText`, `CharNGram`, in the `vocab` module, that ease the process of downloading embeddings, and mapping them to our vocabulary. Each of these classes provides different embeddings trained on different datasets and using different techniques. Let's look at some of the different embeddings provided:

- `charngram.100d`
- `fasttext.en.300d`
- `fasttext.simple.300d`
- `glove.42B.300d`
- `glove.840B.300d`
- `glove.twitter.27B.25d`
- `glove.twitter.27B.50d`
- `glove.twitter.27B.100d`
- `glove.twitter.27B.200d`
- `glove.6B.50d`
- `glove.6B.100d`

- `glove.6B.200d`
- `glove.6B.300d`

The `build_vocab` method of the `Field` object takes in an argument for the embeddings. The following code explains how we download the embeddings:

```
from torchtext.vocab import GloVe
TEXT.build_vocab(train, vectors=GloVe(name='6B', dim=300),max_size=10000,min_
LABEL.build_vocab(train,)
```

The value to the argument vector denotes what embedding class is to be used. The `name` and `dim` arguments determine on what embeddings can be used. We can easily access the embeddings from the `vocab` object. The following code demonstrates it, along with a view of how the results will look:

```
TEXT.vocab.vectors

#Output
0.0000 0.0000 0.0000 ... 0.0000 0.0000 0.0000
 0.0000 0.0000 0.0000 ... 0.0000 0.0000 0.0000
 0.0466 0.2132 -0.0074 ... 0.0091 -0.2099 0.0539
          ...   ⋱   ...
 0.0000 0.0000 0.0000 ... 0.0000 0.0000 0.0000
 0.7724 -0.1800 0.2072 ... 0.6736 0.2263 -0.2919
 0.0000 0.0000 0.0000 ... 0.0000 0.0000 0.0000
[torch.FloatTensor of size 10002x300]
```

Now we have downloaded and mapped the embeddings to our vocabulary. Let's understand how we can use them with a PyTorch model.

Loading the embeddings in the model

The `vectors` variable returns a torch tensor of shape `vocab_size x dimensions` containing the pretrained embeddings. We have to store the embeddings to the weights of our embedding layer. We can assign the weights of the embeddings by accessing the weights of the embeddings layer as demonstrated by the following code.

```
model.embedding.weight.data = TEXT.vocab.vectors
```

`model` represents the object of our network, and `embedding` represents the embedding layer. As we are using the embedding layer with new dimensions, there will be a small change in the input to the linear layer that comes after the embedding layer. The following code has the new architecture, which is similar to the previously-used architecture where we trained our embeddings:

```
class EmbNet(nn.Module):
    def __init__(self,emb_size,hidden_size1,hidden_size2=400):
        super().__init__()
        self.embedding = nn.Embedding(emb_size,hidden_size1)
        self.fc1 = nn.Linear(hidden_size2,3)


    def forward(self,x):
        embeds = self.embedding(x).view(x.size(0),-1)
        out = self.fc1(embeds)
        return F.log_softmax(out,dim=-1)

model = EmbNet(len(TEXT.vocab.stoi),300,12000)
```

Once the embeddings are loaded, we have to ensure that, during training, we do not change the embedding weights. Let's discuss how to achieve that.

Freeze the embedding layer weights

It is a two-step process to tell PyTorch not to change the weights of the embedding layer:

1. Set the `requires_grad` attribute to `False`, which instructs PyTorch that it does not need gradients for these weights.
2. Remove the passing of the embedding layer parameters to the optimizer. If this step is not done, then the optimizer throws an error, as it expects all the parameters to have gradients.

The following code demonstrates how easy it is to freeze the embedding layer weights and instruct the optimizer not to use those parameters:

```
model.embedding.weight.requires_grad = False
optimizer = optim.SGD([ param for param in model.parameters() if param.requir
```

We generally pass all the model parameters to the optimizer, but in the previous code we pass parameters which have `requires_grad` to be `True`.

We can train the model using this exact code and should achieve similar accuracy. All these model architectures fail to take advantage of the sequential nature of the text. In the next section, we explore two popular techniques, namely RNN and Conv1D, that take advantage of the sequential nature of the data.

Recursive neural networks

RNNs are among the most powerful models that enable us to take on applications such as classification, labeling on sequential data, generating sequences of text (such as with the *SwiftKey Keyboard* app which predicts the next word), and converting one sequence to another such as translating a language, say, from French to English. Most of the model architectures such as feedforward neural networks do not take advantage of the sequential nature of data. For example, we need the data to present the features of each example in a vector, say all the tokens that represent a sentence, paragraph, or documents. Feedforward networks are designed just to look at all the features once and map them to output. Let's look at a text example which shows why the order, or sequential nature, is important of text. *I had cleaned my car* and *I had my car cleaned* are two English sentences with the same set of words, but they mean different things only when we consider the order of the words.

Humans make sense of text data by reading words from left to right and building a powerful model that kind of understands all the different things the text says. RNN works slightly similarly, by looking at one word in text at a time. RNN is also a neural network which has a special layer in it, which loops over the data instead of processing all at once. As RNNs can process data in sequence, we can use vectors of different lengths and generate outputs of different lengths. Some of the different representations are provided in the following image:

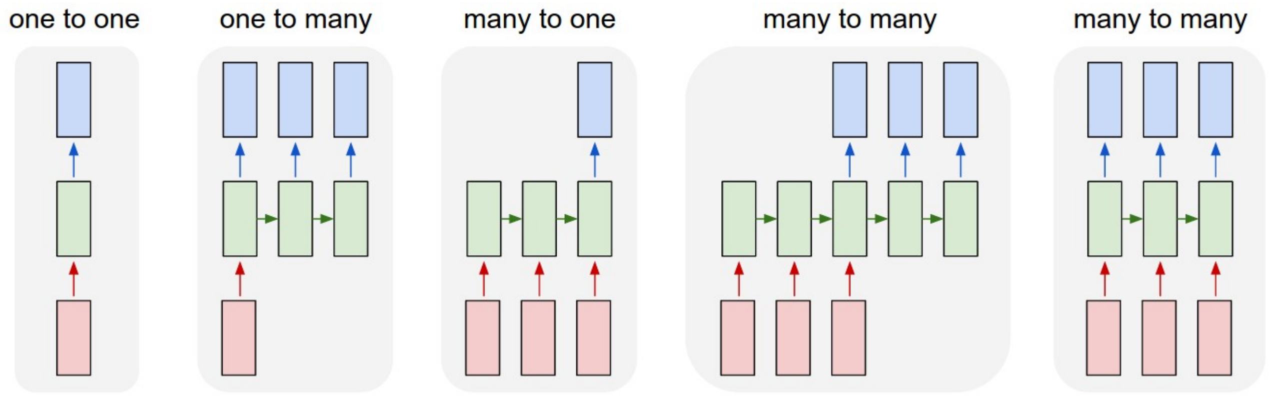

one to one
one to many
many to one
many to many
many to many

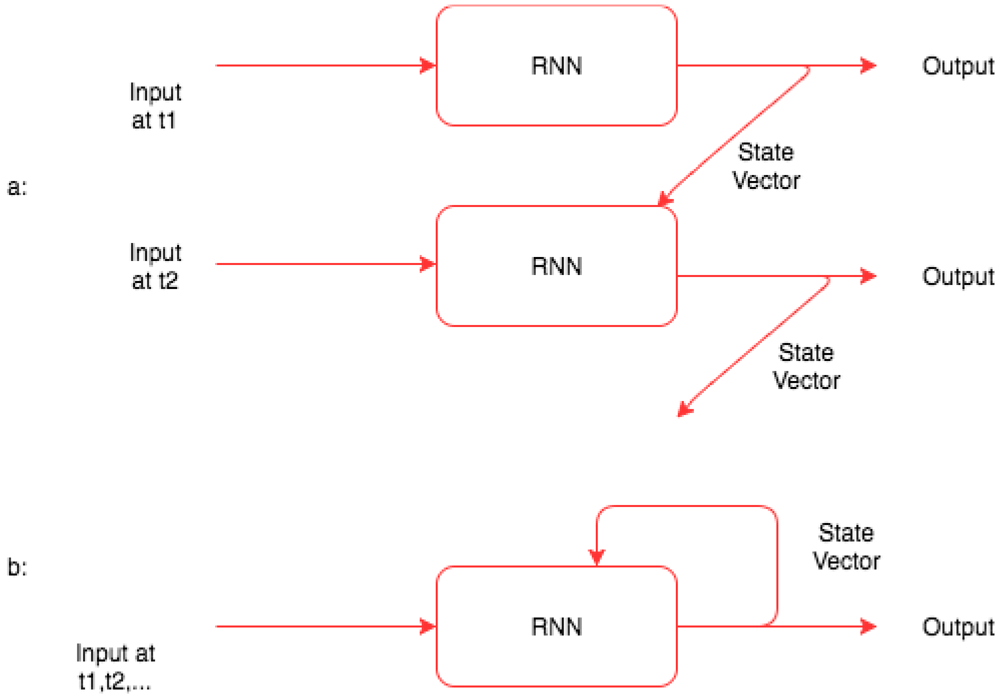
Image source: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

The previous image is from one of the famous blogs on RNN (`http://karpathy.github.io/2015/05/21/rnn-effectiveness`) , in which the author, Andrej Karpathy, writes about how to build an RNN from scratch using Python and use it as sequence generator.

Understanding how RNN works with an example

Let's start with an assumption that we have an RNN model already built, and try to understand what functionality it provides. Once we understand what an RNN does, then let's explore what happens inside an RNN.

Let's consider the Thor review as input to the RNN model. The example text we are looking at is *the action scenes were top notch in this movie....* . We first start by passing the first word, **the**, to our model; the model generates two different things, a **State Vector** and an **Output** vector. The state vector is passed to the model when it processes the next word in the review, and a new state vector is generated. We just consider the **Output** of the model generated during the last sequence. The following figure summarizes it:


a:
Input at t1
RNN
Output
State Vector
Input at t2
RNN
Output
State Vector
b:
State Vector
Input at t1,t2,...
RNN
Output

The preceding figure demonstrates the following:

- How RNN works by unfolding it and the image
- How the state is recursively passed to the same model

By now you will have an idea of what RNN does, but not how it works. Before we get into how it works, let's look at a code snippet which showcases in more detail what we have learnt. We will still view RNN as a black box:

```
rnn = RNN(input_size, hidden_size,output_size)
for i in range(len(Thor_review):
        output, hidden = rnn(thor_review[i], hidden)
```

In the preceding code, the `hidden` variable represents the state vector, sometimes called **hidden state**. By now we should have an idea of how RNN is used. Now, let's look at the code that implements RNN and understand what happens inside the RNN. The following code contains the `RNN` class:

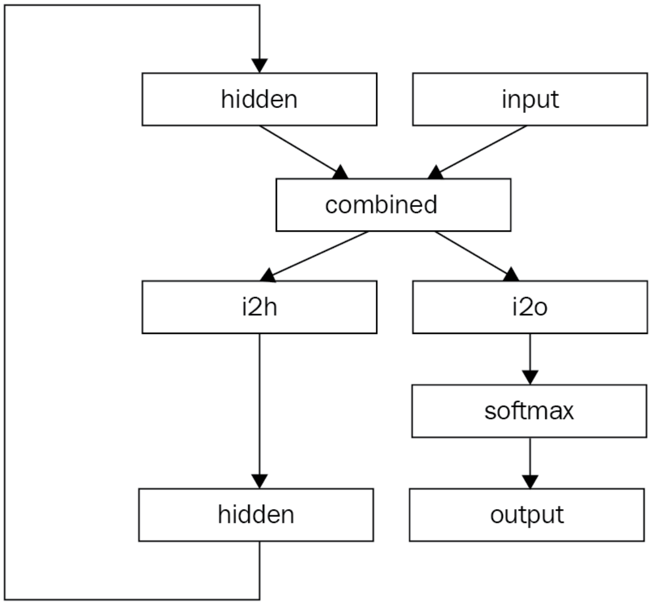
```
import torch.nn as nn
from torch.autograd import Variable

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return Variable(torch.zeros(1, self.hidden_size))
```

Except for the word `RNN` in the preceding code, everything else would sound pretty similar to what we have used in the previous chapters, as PyTorch hides a lot of complexity of backpropagation. Let's walk through the `init` function and the `forward` function to understand what is happening.

The `__init__` function initializes two linear layers, one for calculating the output and the other for calculating the state or hidden vector.

The `forward` function combines the `input` vector and the `hidden` vector and passes it through the two linear layers, which generates an output vector and a hidden state. For the `output` layer, we apply a `log_softmax` function.

The `initHidden` function helps in creating hidden vectors with no state for calling RNN the very first time. Let's take a visual look into what the `RNN` class does in the following figure:



The preceding figure shows how an RNN works.

> *The concepts of RNN are sometimes tricky to understand when you meet them for the first time, so I would strongly recommend some of the amazing blogs provided in the following links:* http://karpathy.github.io/2015/05/21/rnn-effectiveness/ *and* http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

In the next section, we will learn how to use a variant of RNN called **LSTM** to build a sentiment classifier on the `IMDB` dataset.

LSTM

RNNs are quite popular in building real-world applications like language translation, text classification and many more sequential problems, but in reality, we rarely would use a vanilla version of RNN which we saw in the previous section. The vanilla version of RNN has problems like vanishing gradients and gradient explosion when dealing with large sequences. In most of the real-world problems, variants of RNN such as LSTM or GRU are used, which solve the limitations of plain RNN and also have the ability to handle sequential data better. We will try to understand what happens in LSTM, and build a network based on LSTM to solve the text classification problem on the `IMDB` datasets.

Long-term dependency

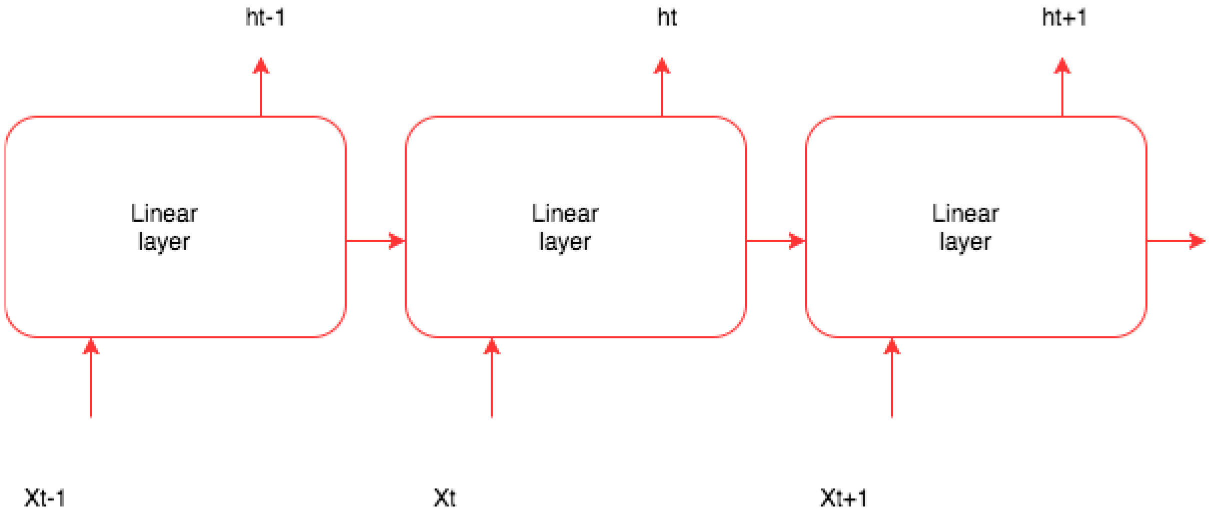
RNNs, in theory, should learn all the dependency required from the historical data to build a context of what happens next. Say, for example, we are trying to predict the last word in the sentence *the clouds are in the sky*. RNN would be able to predict it, as the information (clouds) is just a few words behind. Let's take another long paragraph where the dependency need not be that close, and we want to predict the last word in it. The sentence looks like *I am born in Chennai a city in Tamilnadu. Did schooling in different states of India and I speak...* . The vanilla version of RNN, in practice, finds it difficult to remember the contexts that happened in the earlier parts of sequences. LSTMs and other different variants of RNN solve this problem by adding different neural networks inside the LSTM which later decides how much, or what data can be remembered.

LSTM networks

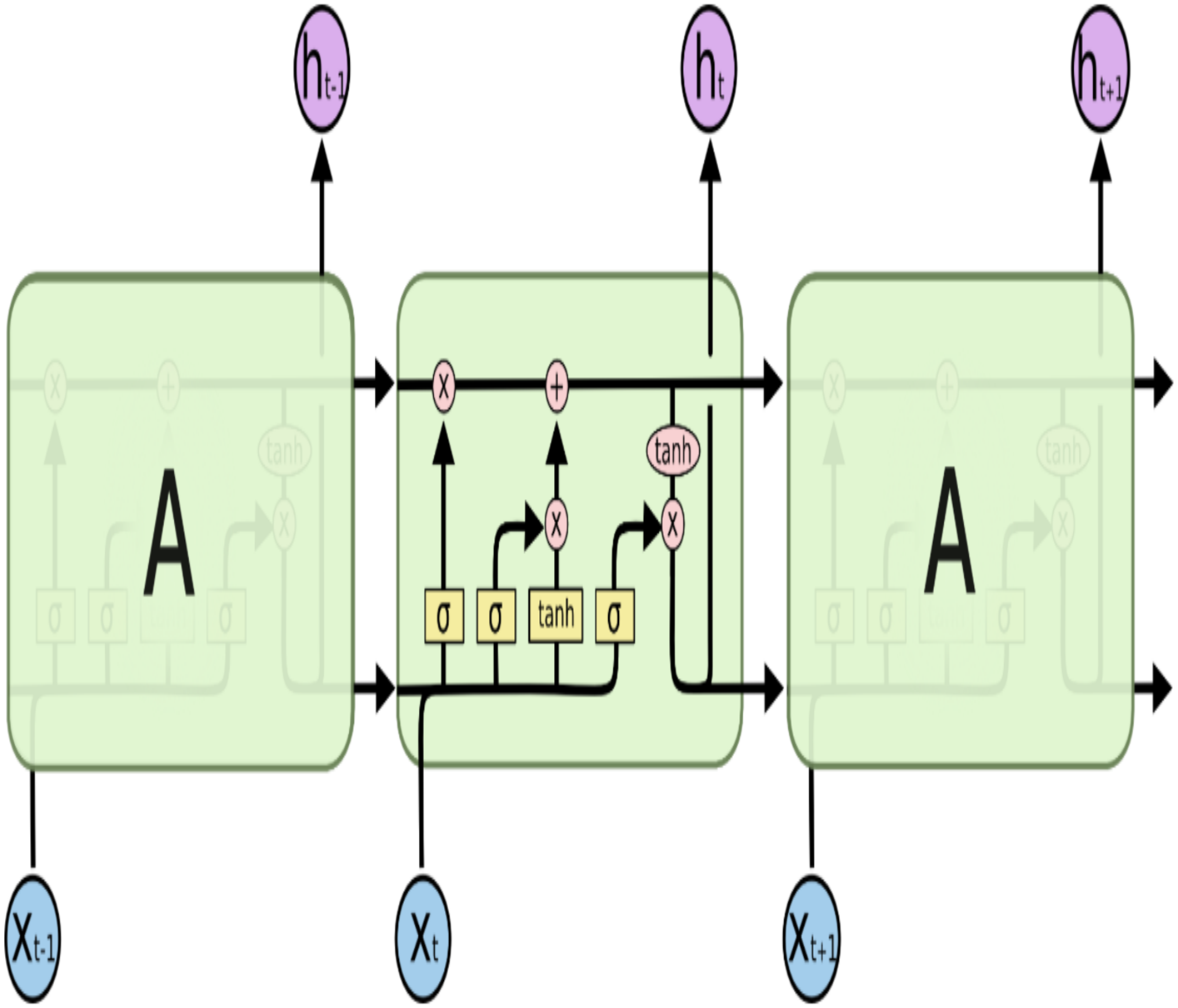
LSTMs are a special kind of RNN, capable of learning long-term dependency. They were introduced in 1997 and got popular in the last few years with advancements in available data and hardware. They work tremendously well on a large variety of problems and are widely used.

LSTMs are designed to avoid long-term dependency problems by having a design by which it is natural to remember information for a long period of time. In RNNs, we saw how they repeat themselves over each element of the sequence. In standard RNNs, the repeating module will have a simple structure like a single linear layer.

The following figure shows how a simple RNN repeats itself:



Inside LSTM, instead of using a simple linear layer we have smaller networks inside the LSTM which does an independent job. The following diagram showcases what happens inside an LSTM:



The repeating module in an LSTM contains four interacting layers.

Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (diagram by Christopher Olah)

Each of the small rectangular (yellow) boxes in the second box in the preceding diagram represents a PyTorch layer, the circles represent an element matrix or vector addition, and the merging lines represent that two vectors are being concatenated. The good part is, we need not implement all of this manually. Most of the modern deep learning frameworks provide an abstraction which will take care of what happens inside an LSTM. PyTorch provides abstraction of all the functionality inside `nn.LSTM` layer, which we can use like any other layer. The most important thing in the LSTM is the cell state that passes through all the iterations, represented by the horizontal line across the cells in the preceding diagram. Multiple networks inside LSTM control what information travels across the cell state. The first step in LSTM (a small network represented by the symbol $\boldsymbol{\sigma}$) is to decide what information is going to be thrown away from the cell state. This network is called **forget gate** and has a sigmoid as an activation function, which outputs values between 0 and 1 for each element in the cell state.The network (PyTorch layer) is represented using the following formula:

$$f_t = \sigma(W_f . [h_{t-1}, X_t] + b_f)$$

The values from the network decide which values are to be held in the cell state and which are to be thrown away. The next step is to decide what information we are going to add to the cell state. This has two parts; a sigmoid layer, called **input gate**, which decides what values to be updated; and a tanh layer, which creates new values to be added to the cell state. The mathematical representation looks like this:

$$i_t \sigma(W_i . [h_{t-1}, X_t] + b_i)$$

$$\acute{C}_t = tanh(W_{\acute{C}} . [h_{t-1}, X_t] + b_{\acute{C}})$$

In the next step, we combine the two values generated by the input gate and tanh. Now we can update the cell state, by doing an element-wise multiplication between the forget gate and the sum of the product of i_t and C_t, as represented by the following formula:

$$C_t = f_t * C_t + i_t * \acute{C}_t$$

Finally, we need to decide on the output, which will be a filtered version of the cell state. There are different versions of LSTM available and most of them work on similar principles. As developers or data scientists, we rarely need to worry about what goes on inside LSTM. If you want to learn more about them, go through the following blog links, which cover a lot of theory in a very intuitive way.

Look at Christopher Olah's amazing blog on LSTM (`http://colah.github.io/posts/2015-08-Understanding-LSTMs`), and another blog from Brandon Rohrer (`https://brohrer.github.io/how_rnns_lstm_work.html`) where he explains LSTM in a nice video.

Since we understand LSTM, let's implement a PyTorch network which we can use to build a sentiment classifier. As usual, we will follow these steps for creating the classifier:

1. Preparing the data
2. Creating the batches
3. Creating the network
4. Training the model

Preparing the data

We use the same torchtext for downloading, tokenizing and building vocabulary for the `IMDB` dataset. When creating the `Field` object, we leave the `batch_first` argument at `False`. RNN networks expect the data to be in the form of `Sequence_length`, `batch_size` and features. The following is used for preparing the dataset:

```
TEXT = data.Field(lower=True,fix_length=200,batch_first=False)
LABEL = data.Field(sequential=False,)
train, test = IMDB.splits(TEXT, LABEL)
TEXT.build_vocab(train, vectors=GloVe(name='6B', dim=300),max_size=10000,min_
LABEL.build_vocab(train,)
```

Creating batches

We use the torchtext `BucketIterator` for creating batches, and the size of the batches will be sequence length and batches. For our case, the size will be [`200`, `32`], where *200* is the sequence length and *32* is the batch size.

The following is the code used for batching:

```
train_iter, test_iter = data.BucketIterator.splits((train, test), batch_size=
train_iter.repeat = False
test_iter.repeat = False
```

Creating the network

Let's look at the code and then walk through the code. You may be surprised at how similar the code looks:

```
class IMDBRnn(nn.Module):

    def __init__(self,vocab,hidden_size,n_cat,bs=1,nl=2):
        super().__init__()
        self.hidden_size = hidden_size
        self.bs = bs
        self.nl = nl
        self.e = nn.Embedding(n_vocab,hidden_size)
        self.rnn = nn.LSTM(hidden_size,hidden_size,nl)
        self.fc2 = nn.Linear(hidden_size,n_cat)
        self.softmax = nn.LogSoftmax(dim=-1)

    def forward(self,inp):
        bs = inp.size()[1]
        if bs != self.bs:
            self.bs = bs
        e_out = self.e(inp)
        h0 = c0 = Variable(e_out.data.new(*(self.nl,self.bs,self.hidden_size)
        rnn_o,_ = self.rnn(e_out,(h0,c0))
        rnn_o = rnn_o[-1]
        fc = F.dropout(self.fc2(rnn_o),p=0.8)
        return self.softmax(fc)
```

The `init` method creates an embedding layer of the size of the vocabulary and `hidden_size`. It also creates an LSTM and a linear layer. The last layer is a `LogSoftmax` layer for converting the results from the linear layer to probabilities.

In the `forward` function, we pass the input data of size [`200`, `32`], which gets passed through the embedding layer and each token in the batch gets replaced by embedding and the size turns to [200, 32, 100], where *100* is the embedding dimensions. The LSTM layer takes the output of the embedding layer along with two hidden variables. The hidden variables should be of the same type of the embeddings output, and their size should be [`num_layers`, `batch_size`, `hidden_size`]. The LSTM processes the data in a sequence and generates the output of the shape [`Sequence_length`, `batch_size`, `hidden_size`], where each sequence index represents the output of that sequence. In this case, we just take the output of the last sequence, which is of shape [`batch_size`, `hidden_dim`], and pass it on to a linear layer to map it to the output categories. Since the model tends to overfit, add a dropout layer. You can play with the dropout probabilities.

Training the model

Once the network is created, we can train the model using the same code as seen in the previous examples. The following is the code for training the model:

```
model = IMDBRnn(n_vocab,n_hidden,3,bs=32)
model = model.cuda()

optimizer = optim.Adam(model.parameters(),lr=1e-3)

def fit(epoch,model,data_loader,phase='training',volatile=False):
    if phase == 'training':
        model.train()
    if phase == 'validation':
        model.eval()
        volatile=True
    running_loss = 0.0
    running_correct = 0
    for batch_idx , batch in enumerate(data_loader):
        text , target = batch.text , batch.label
        if is_cuda:
            text,target = text.cuda(),target.cuda()

        if phase == 'training':
            optimizer.zero_grad()
        output = model(text)
        loss = F.nll_loss(output,target)

        running_loss += F.nll_loss(output,target,size_average=False).data[0]
        preds = output.data.max(dim=1,keepdim=True)[1]
        running_correct += preds.eq(target.data.view_as(preds)).cpu().sum()
        if phase == 'training':
            loss.backward()
            optimizer.step()

    loss = running_loss/len(data_loader.dataset)
    accuracy = 100. * running_correct/len(data_loader.dataset)
    print(f'{phase} loss is {loss:{5}.{2}} and {phase} accuracy is {running_c
    return loss,accuracy

train_losses , train_accuracy = [],[]
val_losses , val_accuracy = [],[]

for epoch in range(1,5):

    epoch_loss, epoch_accuracy = fit(epoch,model,train_iter,phase='training')
    val_epoch_loss , val_epoch_accuracy = fit(epoch,model,test_iter,phase='va
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    val_losses.append(val_epoch_loss)
    val_accuracy.append(val_epoch_accuracy)
```

Following is the result of the training model:

```
#Results

training loss is   0.7 and training accuracy is 12564/25000     50.26
validation loss is   0.7 and validation accuracy is 12500/25000      50.0
training loss is  0.66 and training accuracy is 14931/25000     59.72
validation loss is  0.57 and validation accuracy is 17766/25000     71.06
training loss is  0.43 and training accuracy is 20229/25000     80.92
validation loss is   0.4 and validation accuracy is 20446/25000     81.78
training loss is   0.3 and training accuracy is 22026/25000      88.1
validation loss is  0.37 and validation accuracy is 21009/25000     84.04
```

Training the model for four epochs gave an accuracy of 84%. Training for more epochs resulted in an overfitted model, as the loss started increasing. We can try some of the techniques that we tried such as decreasing the hidden dimensions, increasing sequence length, and training with smaller learning rates to further improve the accuracy.

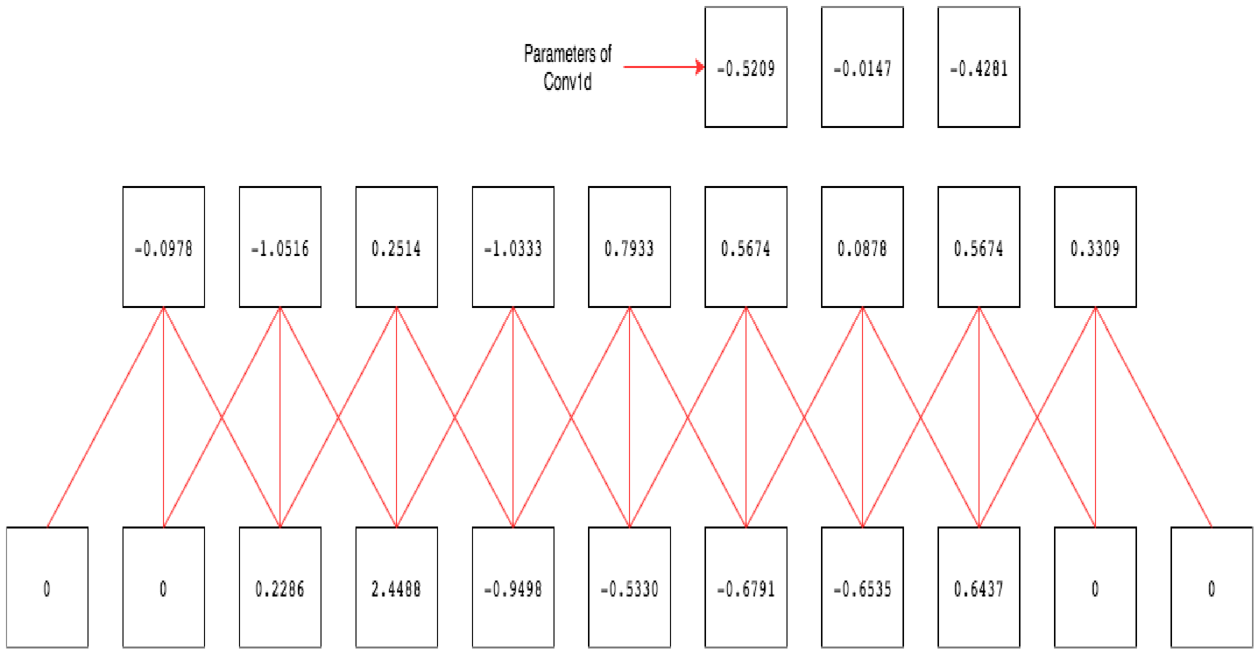
We will also explore how we can use one-dimensional convolutions for training on sequence data.

Convolutional network on sequence data

We learned how CNNs solve problems in computer vision by learning features from the images. In images, CNNs work by convolving across height and width. In the same way, time can be treated as a convolutional feature. One-dimensional convolutions sometimes perform better than RNNs and are computationally cheaper. In the last few years, companies like Facebook have shown success in audio generation and machine translation. In this section, we will learn how CNNs can be used to build a text classification solution.

Understanding one-dimensional convolution for sequence data

In `Chapter 5`, *Deep Learning for Computer Vision*, we have seen how two-dimensional weights are learned from the training data. These weights move across the image to generate different activations. In the same way, one-dimensional convolution activations are learned during training of our text classifier, where these weights learn patterns by moving across the data. The following diagram explains how one-dimensional convolutions will work:


Parameters of Conv1d
-0.5209
-0.0147
-0.4281
-0.0978
-1.0516
0.2514
-1.0333
0.7933
0.5674
0.0878
0.5674
0.3309
0
0
0.2286
2.4488
-0.9498
-0.5330
-0.6791
-0.6535
0.6437
0
0

For training a text classifier on the `IMDB` dataset, we will follow the same steps as we followed for building the classifier using LSTM. The only thing that changes is that we use `batch_first = True`, unlike in our LSTM network. So, let's look at the network, the training code, and its results.

Creating the network

Let's look at the network architecture and then walk through the code:

```
class IMDBCnn(nn.Module):

    def __init__(self,vocab,hidden_size,n_cat,bs=1,kernel_size=3,max_len=200)
        super().__init__()
        self.hidden_size = hidden_size
        self.bs = bs
    self.e = nn.Embedding(n_vocab,hidden_size)
    self.cnn = nn.Conv1d(max_len,hidden_size,kernel_size)
    self.avg = nn.AdaptiveAvgPool1d(10)
        self.fc = nn.Linear(1000,n_cat)
        self.softmax = nn.LogSoftmax(dim=-1)

    def forward(self,inp):
        bs = inp.size()[0]
        if bs != self.bs:
            self.bs = bs
        e_out = self.e(inp)
        cnn_o = self.cnn(e_out)
        cnn_avg = self.avg(cnn_o)
        cnn_avg = cnn_avg.view(self.bs,-1)
        fc = F.dropout(self.fc(cnn_avg),p=0.5)
        return self.softmax(fc)
```

In the preceding code, instead of an LSTM layer we have a `Conv1d` layer and an `AdaptiveAvgPool1d` layer. The convolution layer accepts the sequence length as its input size, and the output size to the hidden size, as the kernel size three. Since we have to change the dimensions of the linear layer, every time we try to run it with different lengths we use an `AdaptiveAvgpool1d` which takes input of any size and generates an output of the given size. So, we can use a linear layer whose size is fixed. The rest of the code is similar to what we have seen in most of the network architectures.

Training the model

The training steps for the model are the same as the previous example. Let's just look at the code to call the `fit` method and the results it generated:

```
train_losses , train_accuracy = [],[]
val_losses , val_accuracy = [],[]

for epoch in range(1,5):

    epoch_loss, epoch_accuracy = fit(epoch,model,train_iter,phase='training')
    val_epoch_loss , val_epoch_accuracy = fit(epoch,model,test_iter,phase='va
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    val_losses.append(val_epoch_loss)
    val_accuracy.append(val_epoch_accuracy)
```

We ran the model for four epochs, which gave approximately 83% accuracy. Here are the results of running the model:

```
training loss is  0.59 and training accuracy is 16724/25000      66.9
validation loss is  0.45 and validation accuracy is 19687/25000     78.75
training loss is  0.38 and training accuracy is 20876/25000      83.5
validation loss is   0.4 and validation accuracy is 20618/25000     82.47
training loss is  0.28 and training accuracy is 22109/25000     88.44
validation loss is  0.41 and validation accuracy is 20713/25000     82.85
training loss is  0.22 and training accuracy is 22820/25000     91.28
validation loss is  0.44 and validation accuracy is 20641/25000     82.56
```

Since the `validation loss` started increasing after three epochs, I stopped running the model. A couple of things that we could try to improve the results are using pretrained weights, adding another convolution layer, and trying a `MaxPool1d` layer between the convolutions. I leave it to you to try this and see if that helps improve the accuracy.

Summary

In this chapter, we learned different techniques to represent text data in deep learning. We learned how to use pretrained word embeddings and our own trained embeddings when working on a different domain. We built a text classifier using LSTMs and one-dimensional convolutions.

In the next chapter, we will learn how to train deep learning algorithms to generate stylish images, and new images, and to generate text.

Generative Networks

All the examples that we have seen in the previous chapters were focused on solving problems such as classification or regression. This chapter is very interesting and important for understanding how deep learning is being evolved to solve problems in unsupervised learning.

In this chapter, we will train networks that learn how to create:

- Images based on content and a particular artistic style, popularly called **style transfer**
- Generating faces of new persons using a particular type of **generative adversarial network** (**GAN**)
- Generating new text using language modeling

These techniques form the basis of most of the advanced research that is happening in the deep learning space. Going into the exact specifics of each of the subfields, such as GANs and language modeling is out of the scope of this book, as they deserve a separate book for themselves. We will learn how they work in general and the process of building them in PyTorch.

Neural style transfer

We humans generate artwork with different levels of accuracy and complexity. Though the process of creating art could be a very complex process, it can be seen as a combination of the two most important factors, namely, what to draw and how to draw. What to draw is inspired by what we see around us, and how we draw will also take influences from certain things that are found around us. This could be an oversimplification from an artist's perspective, but for understanding how we can create artwork using deep learning algorithms, it is very useful. We will train a deep learning algorithm to take content from one image and then draw it according to a specific artistic style. If you are an artist or in the creative industry, you can directly use the amazing research that has gone on in recent years to improve this and create something cool within the domain you work in. Even if you are not, it still introduces you to the field of generative models, where networks generate new content.

Let's understand what is done in neural style transfer at a high-level, and then dive into details, along with the PyTorch code required to build it. The style transfer algorithm is provided with a content *image (C)* and a style *image (S);* the algorithm has to generate a new image (O) which has the content from the content image and the style from the style image. This process of creating neural style transfer was introduced by Leon Gates and others in 2015 (`A Neural Algorithm of Artistic Style`). The following is the content image (C) that we will be using:

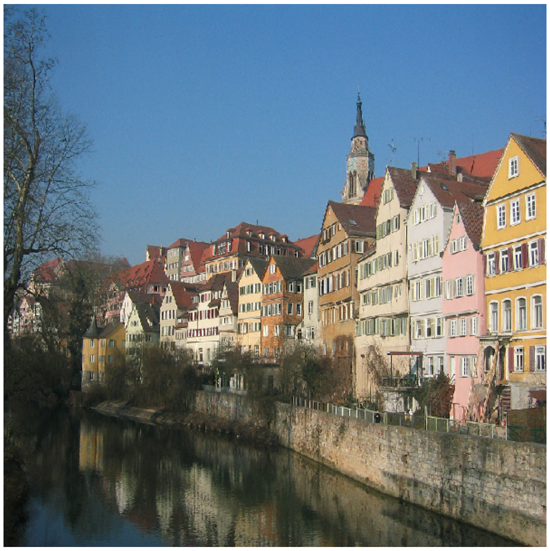


Image source: https://arxiv.org/pdf/1508.06576.pdf

And the following is the style image (S):

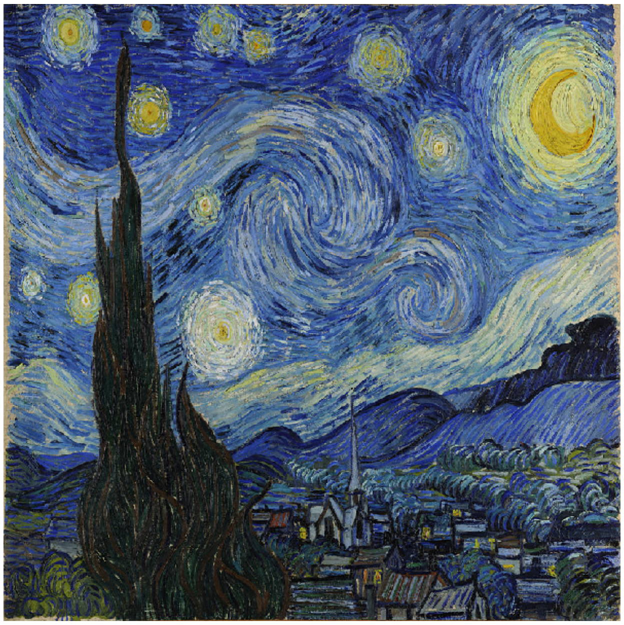


Image source: https://arxiv.org/pdf/1508.06576.pdf

And this is the image that we are going to generate:

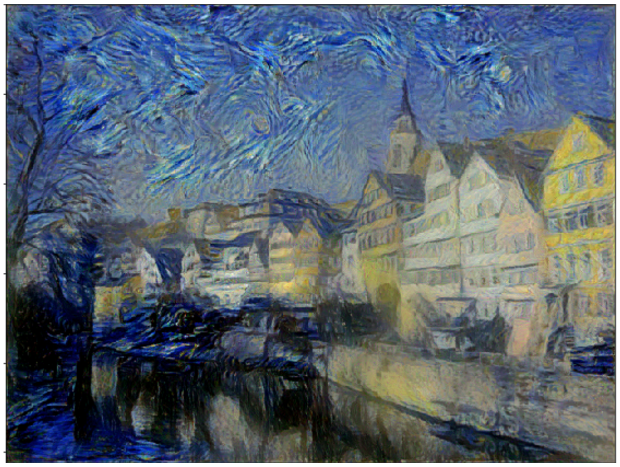


Image source: https://arxiv.org/pdf/1508.06576.pdf

The idea behind style transfer becomes intuitive from understanding how **Convolutional Neural Networks** (**CNNs**) work. When CNNs are trained for object recognition, the early layers of a trained CNN learn very generic information like lines, curves, and shapes. The last layers in a CNN capture the higher-level concepts from an image such as eyes, buildings, and trees. So the values of the last layers of similar images tend to be closer. We take the same concept and apply it for content loss. The last layer for the content image and the generated image should be similar, and we calculate the similarity using **mean square error** (**MSE**). We use our optimization algorithms to bring down the loss value.

The style of the image is generally captured across multiple layers in a CNN by a technique called **gram matrix**. Gram matrix calculates the correlation between the feature maps captured across multiple layers. Gram matrix gives a measure of calculating the style. Similarly styled images have similar values for gram matrix. The style loss is also calculated using MSE between the gram matrix of the style image and the generated image.

We will use a pretrained VGG19 model, provided in the torchvision models. The steps required for training a style transfer model are similar to any other deep learning model, except for the fact that calculating losses is more involved than for a classification or a regression model. The training of the neural style algorithm can be broken down to the following steps:

1. Loading data.
2. Creating a VGG19 model.
3. Defining content loss.
4. Defining style loss.
5. Extracting losses across layers from VGG model.
6. Creating an optimizer.
7. Training—generating an image similar to the content image, and style similar to the style image.

Loading the data

Loading data is similar to what we have seen for solving image classification problems in `Chapter 5`, *Deep Learning for Computer Vision*. We will be using the pretrained VGG model, so we have to normalize the images using the same values on which the pretrained model is trained.

The following code shows how we can do this. The code is mostly self-explanatory as we have already discussed it in detail in the previous chapters:

```
#Fixing the size of the image, reduce it further if you are not using a GPU.
imsize = 512
is_cuda = torch.cuda.is_available()

#Converting image ,making it suitable for training using the VGG model.

prep = transforms.Compose([transforms.Resize(imsize),
                           transforms.ToTensor(),
                           transforms.Lambda(lambda x: x[torch.LongTensor([2,
                           transforms.Normalize(mean=[0.40760392, 0.45795686,
                                                std=[1,1,1]),
                           transforms.Lambda(lambda x: x.mul_(255)),
                          ])

#Converting the generated image back to a format which we can visualise.

postpa = transforms.Compose([transforms.Lambda(lambda x: x.mul_(1./255)),
                           transforms.Normalize(mean=[-0.40760392, -0.4579568
                                                std=[1,1,1]),
                           transforms.Lambda(lambda x: x[torch.LongTensor([2,
                           ])
postpb = transforms.Compose([transforms.ToPILImage()])

#This method ensures data in the image does not cross the permissible range .
```

```
def postp(tensor): # to clip results in the range [0,1]
    t = postpa(tensor)
    t[t>1] = 1
    t[t<0] = 0
    img = postpb(t)
    return img

#A utility function to make data loading easier.
def image_loader(image_name):
    image = Image.open(image_name)
    image = Variable(prep(image))
    # fake batch dimension required to fit network's input dimensions
    image = image.unsqueeze(0)
    return image
```

In this code, we defined three functionalities, `prep` does all the preprocessing required and uses the same values for normalization as those with which the VGG model was trained. The output of the model needs to be normalized back to its original values; the `postpa` function does the processing required. The generated model may be out of the range of accepted values, and the `postp` function limits all the values greater than 1 to 1 and values that are less than 0 to 0. Finally, the `image_loader` function loads the image, applies the preprocessing transformation, and converts it into a variable. The following function loads the style and content image:

```
style_img = image_loader("Images/vangogh_starry_night.jpg")
content_img = image_loader("Images/Tuebingen_Neckarfront.jpg")
```

We can either create an image with noise (random numbers) or we can use the same content image. We will use the content image in this case. The following code creates the content image:

```
opt_img = Variable(content_img.data.clone(),requires_grad=True)
```

We will use an optimizer to tune the values of the `opt_img` in order for the image to be closer to the content image and style image. For that reason, we are asking PyTorch to maintain the gradients by mentioning `requires_grad=True`.

Creating the VGG model

We will load a pretrained model from `torchvisions.models`. We will be using this model only for extracting features, and the PyTorch VGG model is defined in such a way that all the convolutional blocks will be in the `features` module and the fully connected, or linear, layers are in the `classifier` module. Since we will not be training any of the weights or parameters in the VGG model, we will also freeze the model. The following code demonstrates the same:

```
#Creating a pretrained VGG model
vgg = vgg19(pretrained=True).features

#Freezing the layers as we will not use it for training.
for param in vgg.parameters():
    param.requires_grad = False
```

In this code, we created a VGG model, used only its convolution blocks and froze all the parameters of the model, as we will be using it only for extracting features.

Content loss

The **content loss** is the MSE calculated on the output of a particular layer, extracted by passing two images through the network. We extract the outputs of the intermediate layers from the VGG by using the `register_forward_hook` functionality, by passing in the content image and the image to be optimized. We calculate the MSE obtained from the outputs of these layers, as described in the following code:

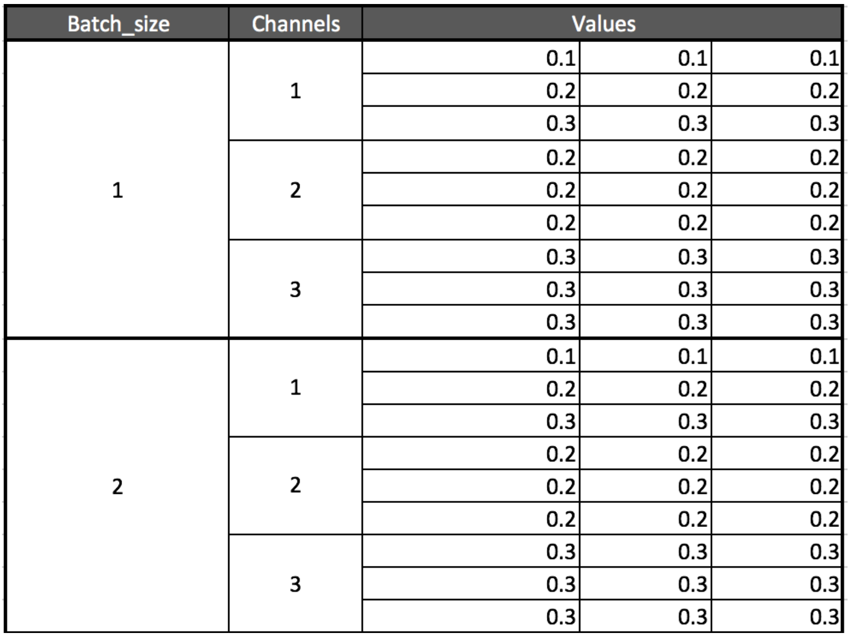
```
target_layer = dummy_fn(content_img)
noise_layer = dummy_fn(noise_img)
criterion = nn.MSELoss()
content_loss = criterion(target_layer,noise_layer)
```

We will implement the `dummy_fn` of this code—in the coming sections. For now, all we know is, that the `dummy_fn` function returns the outputs of particular layers by passing an image. We pass the outputs generated by passing the content image and noise image to the MSE `loss` function.

Style loss

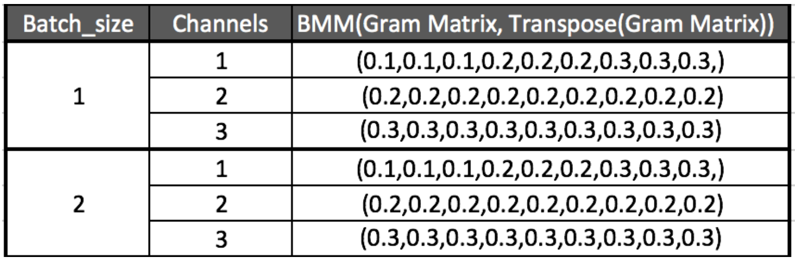
The **style loss** is calculated across multiple layers. Style loss is the MSE of the gram matrix generated for each feature map. The gram matrix represents the correlation value of its features. Let's understand how gram matrix works by using the following diagram and a code implementation.

The following table shows the output of a feature map of dimension [2, 3, 3, 3], having the column attributes `Batch_size`, `Channels`, and `Values`:



Batch_size	Channels	Values		
1	1	0.1	0.1	0.1
		0.2	0.2	0.2
		0.3	0.3	0.3
	2	0.2	0.2	0.2
		0.2	0.2	0.2
		0.2	0.2	0.2
	3	0.3	0.3	0.3
		0.3	0.3	0.3
		0.3	0.3	0.3
2	1	0.1	0.1	0.1
		0.2	0.2	0.2
		0.3	0.3	0.3
	2	0.2	0.2	0.2
		0.2	0.2	0.2
		0.2	0.2	0.2
	3	0.3	0.3	0.3
		0.3	0.3	0.3
		0.3	0.3	0.3

To calculate the gram matrix, we flatten all the values per channel and then find correlation by multiplying with its transpose, as shown in the following table:



Batch_size	Channels	BMM(Gram Matrix, Transpose(Gram Matrix))
1	1	(0.1,0.1,0.1,0.2,0.2,0.2,0.3,0.3,0.3,)
	2	(0.2,0.2,0.2,0.2,0.2,0.2,0.2,0.2,0.2)
	3	(0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3)
2	1	(0.1,0.1,0.1,0.2,0.2,0.2,0.3,0.3,0.3,)
	2	(0.2,0.2,0.2,0.2,0.2,0.2,0.2,0.2,0.2)
	3	(0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3)

All we did is flatten all the values, with respect to each channel, to a single vector or tensor. The following code implements this:

```
class GramMatrix(nn.Module):

    def forward(self,input):
        b,c,h,w = input.size()
        features = input.view(b,c,h*w)
        gram_matrix = torch.bmm(features,features.transpose(1,2))
        gram_matrix.div_(h*w)
        return gram_matrix
```

We implement the `GramMatrix` as another PyTorch module with a `forward` function so that we can use it like a PyTorch layer. We are extracting the different dimensions from the input image in this line:

```
b,c,h,w = input.size()
```

Here, `b` represents batch, `c` represents filters or channels, `h` represents height, and `w` represents width. In the next step, we will use the following code to keep the batch and channel dimensions intact and flatten all the values along the height and width dimension, as shown in the preceding figure:

```
features = input.view(b,c,h*w)
```

The gram matrix is calculated by multiplying the flattening values along with its transposed vector. We can do it by using the PyTorch batch matrix multiplication function, provided as `torch.bmm()`, as shown in the following code:

```
gram_matrix = torch.bmm(features,features.transpose(1,2))
```

We finish normalizing the values of the gram matrix by dividing it by the number of elements. This prevents a particular feature map with a lot of values dominating the score. Once `GramMatrix` is calculated, it becomes simple to calculate style loss, which is implemented in this code:

```
class StyleLoss(nn.Module):

    def forward(self,inputs,targets):
        out = nn.MSELoss()(GramMatrix()(inputs),targets)
        return (out)
```

The `StyleLoss` is implemented as another PyTorch layer. It calculates the MSE between the input `GramMatrix` values and the style image `GramMatrix` values.

Extracting the losses

Just like we extracted the activation of a convolution layer using the `register_forward_hook()` function in Chapter 5, *Deep Learning for Computer Vision,* we can extract losses of different convolutional layers required to calculate style loss and content loss. The one difference in this case is that instead of extracting from one layer, we need to extract outputs of multiple layers. The following class integrates the required change:

```
class LayerActivations():
    features=[]

    def __init__(self,model,layer_nums):

        self.hooks = []
        for layer_num in layer_nums:
            self.hooks.append(model[layer_num].register_forward_hook(self.hoc

    def hook_fn(self,module,input,output):
        self.features.append(output)

    def remove(self):
        for hook in self.hooks:
            hook.remove()
```

The `__init__` method takes the model on which we need to call the `register_forward_hook` method and the layer numbers for which we need to extract the outputs. The `for` loop in the `__init__` method iterates through the layer numbers and registers the forward hook required to pull the outputs.

The `hook_fn` passed to the `register_forward_hook` method is called by PyTorch after that layer on which the `hook_fn` function is registered. Inside the function, we capture the output and store it in the `features` array.

We need to call the `remove` function once when we don't want to capture the outputs. Forgetting to invoke the `remove` methods can cause out-of-memory exceptions as all the outputs get accumulated.

Let's write another utility function which can extract the outputs required for style and content images. The following function does the same:

```
def extract_layers(layers,img,model=None):


    la = LayerActivations(model,layers)
    #Clearing the cache
    la.features = []
    out = model(img)
    la.remove()
    return la.features
```

Inside the `extract_layers` function, we create objects for the `LayerActivations` class by passing in the model and the layer numbers. The features list may contain outputs from previous runs, so we are reinitiating to an empty list. Then we pass in the image through the model, and we are not going to use the outputs. We are more interested in the outputs generated in the `features` array. We call the `remove` method to remove all the registered hooks from the model and return the features. The following code shows how we extract the targets required for style and content image:

```
content_targets = extract_layers(content_layers,content_img,model=vgg)
style_targets = extract_layers(style_layers,style_img,model=vgg)
```

Once we extract the targets, we need to detach the outputs from the graphs that created them. Remember that all these outputs are PyTorch variables which maintain information of how they are created. But, for our case, we are interested in only the output values and not the graph, as we are not going to update either `style` image or the `content` image. The following code illustrates this technique:

```
content_targets = [t.detach() for t in content_targets]
style_targets = [GramMatrix()(t).detach() for t in style_targets]
```

Once we have detached, let's add all the targets into one list. The following code illustrates this technique:

```
targets = style_targets + content_targets
```

When calculating the style loss and content loss, we passed on two lists called content layers and style layers. Different layer choices will have an impact on the quality of the image generated. Let's pick the same layers as the authors of the paper have mentioned. The following code shows the choice of layers that we are using here:

```
style_layers = [1,6,11,20,25]
content_layers = [21]
loss_layers = style_layers + content_layers
```

The optimizer expects a single scalar quantity to minimize. To achieve a single scalar value, we sum up all the losses that have arrived at different layers. It is common practice to do a weighted sum of these losses, and again we pick the same weights as used in the paper's implementation in the GitHub repository (`https://github.com/leongatys/PytorchNeuralStyleTransfer`). Our implementation is a slightly modified version of the author's implementation. The following code describes the weights being used, which are calculated by the number of filters in the selected layers:

```
style_weights = [1e3/n**2 for n in [64,128,256,512,512]]
content_weights = [1e0]
weights = style_weights + content_weights
```

To visualize this, we can print the VGG layers. Take a minute to observe what layers we are picking, and you can experiment with different layer combinations. We will use the following code to `print` the VGG layers:

```
print(vgg)

#Results

Sequential(
  (0): Conv2d (3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace)
  (2): Conv2d (64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace)
  (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (5): Conv2d (64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace)
  (7): Conv2d (128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace)
  (9): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (10): Conv2d (128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace)
  (12): Conv2d (256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace)
  (14): Conv2d (256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace)
  (16): Conv2d (256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (17): ReLU(inplace)
  (18): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (19): Conv2d (256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace)
  (21): Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace)
  (23): Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (24): ReLU(inplace)
  (25): Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (26): ReLU(inplace)
  (27): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (28): Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (29): ReLU(inplace)
  (30): Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (31): ReLU(inplace)
  (32): Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (33): ReLU(inplace)
  (34): Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (35): ReLU(inplace)
  (36): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
)
```

We have to define the `loss` functions and the `optimizer` to generate artistic images. We will initialize both of them in the following section.

Creating loss function for each layers

We have already defined `loss` functions as PyTorch layers. So, let's create the loss layers for different style losses and content losses. The following code defines the function:

```
loss_fns = [StyleLoss()] * len(style_layers) + [nn.MSELoss()] * len(content_l
```

The `loss_fns` is a list containing a bunch of style loss objects and content loss objects based on the lengths of the arrays created.

Creating the optimizer

In general, we pass in the parameters of a network like VGG to be trained. But, in this example, we are using VGG models as feature extractors and, hence, we cannot pass the VGG parameters. Here, we will only provide the parameters of the `opt_img` variable that we will optimize to make the image have the required content and style. The following code creates the `optimizer` that optimizes its values:

```
optimizer = optim.LBFGS([opt_img]);
```

Now we have all the components for training.

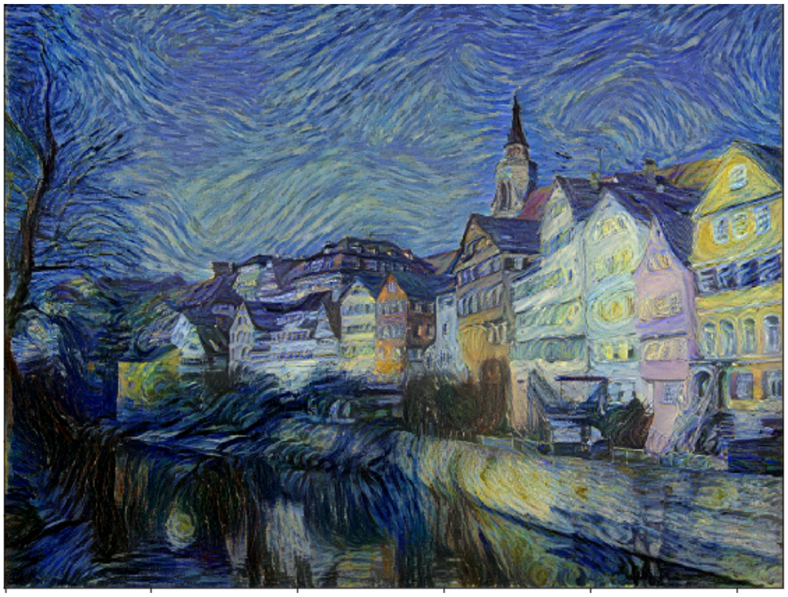
Training

The `training` method is different compared to the other models that we have trained till now. Here, we need to calculate loss at multiple layers, and every time the optimizer is called, it will change the input image so that its content and style gets close to the target's content and style. Let's look at the code used for training, and then we will walk through the important steps in the training:

```
max_iter = 500
show_iter = 50
n_iter=[0]

while n_iter[0] <= max_iter:

    def closure():
        optimizer.zero_grad()

        out = extract_layers(loss_layers,opt_img,model=vgg)
        layer_losses = [weights[a] * loss_fns[a](A, targets[a]) for a,A in er
        loss = sum(layer_losses)
        loss.backward()
        n_iter[0]+=1
        #print loss
        if n_iter[0]%show_iter == (show_iter-1):
            print('Iteration: %d, loss: %f'%(n_iter[0]+1, loss.data[0]))

        return loss

    optimizer.step(closure)
```

We are running the training loop for `500` iterations. For every iteration, we calculate the output from different layers of the VGG model using our `extract_layers` function. In this case, the only thing that changes is the values of `opt_img`, which will contain our style image. Once the outputs are calculated, we are calculating the losses by iterating through the outputs and passing them to the corresponding `loss` functions along with their respective targets. We sum up all the losses and call the `backward` function. At the end of the `closure` function, loss is returned. The `closure` method is called along with the `optimizer.step` method for `max_iter`. If you are running on a GPU, it could take a few minutes to run; if you are running on a CPU, try reducing the size of the image to make it run faster.

After running for 500 epochs, the resulting image on my machine looks as shown here. Try different combinations of content and style to generate interesting images:



In the next section, let's go ahead and generate human faces using **deep convolutional generative adversarial networks** (**DCGANs**).

Generative adversarial networks

GANs have become very popular in the last few years. Every week there are some advancements being made in the area of GANs. It has become one of the important subfields of deep learning, with a very active research community. GAN was introduced by Ian Goodfellow in 2014. The GAN addresses the problem of unsupervised learning by training two deep neural networks, called *generator* and *discriminator*, which compete with each other. In the course of training, both eventually become better at the tasks that they perform.

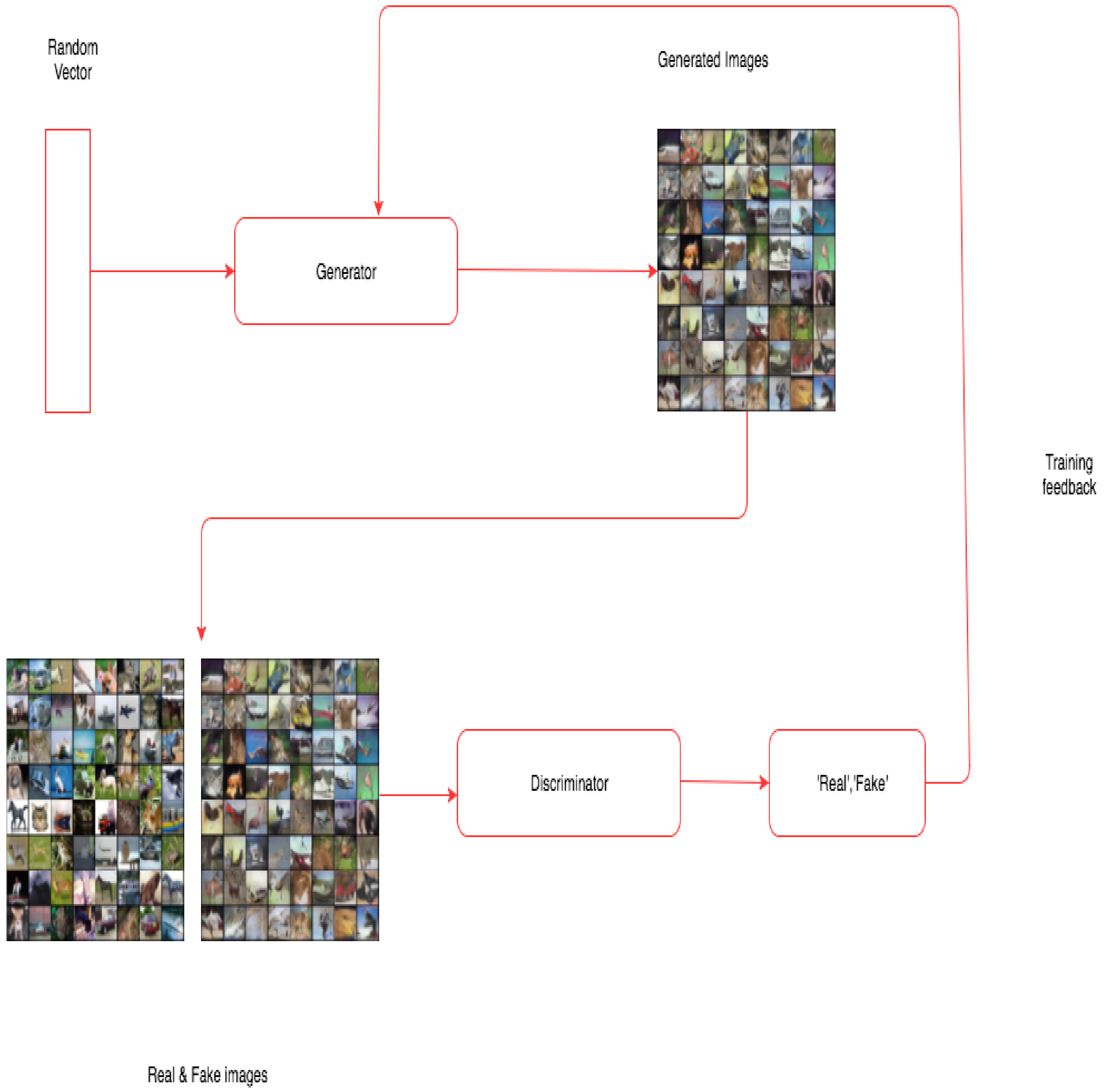
GANs are intuitively understood using the case of counterfeiter (generator) and the police (discriminator). Initially, the counterfeiter shows the police fake money. The police identifies it as fake and explains to the counterfeiter why it is fake. The counterfeiter makes new fake money based on the feedback it received. The police finds it fake and informs the counterfeiter why it is fake. It repeats this a huge number of times until the counterfeiter is able to make fake money which the police is unable to recognize. In the GAN scenario, we end up with a generator that generates fake images which are quite similar to the real ones, and a classifier becomes great at identifying a fake from the real thing.

GAN is a combination of a forger network and an expert network, each being trained to beat the other. The generator network takes a random vector as input and generates a synthetic image. The discriminator network takes an input image and predicts whether the image is real or fake. We pass the discriminator network either a real image or a fake image.

The generator network is trained to produce images and fool the discriminator network into believing they are real. The discriminator network is also constantly improving at not getting fooled, as we pass the feedback whilst training it. Though the idea of GANs sounds simple in theory, training a GAN model that actually works is very difficult. Training a GAN is also challenging, as there are two deep neural networks that need to be trained.

> *The DCGAN is one of the early models that demonstrated how to build a GAN model that learns by itself and generates meaningful images. You can learn more about it here:*
>
> `https://arxiv.org/pdf/1511.06434.pdf`

The following diagram shows the architecture of a GAN model:


Random Vector
Generator
Generated Images
Training feedback
Discriminator
'Real','Fake'
Real & Fake images

We will walk through each of the components of this architecture, and some of the reasoning behind them, and then we will implement the same flow in PyTorch in the next section. By the end of this implementation, we will have basic knowledge of how DCGANs work.

Deep convolutional GAN

In this section, we will implement different parts of training a GAN architecture, based on the *DCGAN paper I* mentioned in the preceding information box. Some of the important parts of training a DCGAN include:

- A generator network, which maps a latent vector (list of numbers) of some fixed dimension to images of some shape. In our implementation, the shape is (3, 64, 64).
- A discriminator network, which takes as input an image generated by the generator or from the actual dataset, and maps to that a score estimating if the input image is real or fake.
- Defining loss functions for generator and discriminator.
- Defining an optimizer.
- Training a GAN.

Let's explore each of these sections in detail. The implementation is based on the code, which is available in the PyTorch examples at:

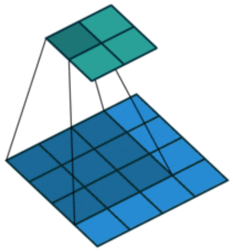
`https://github.com/pytorch/examples/tree/master/dcgan`

Defining the generator network

The generator network takes a random vector of fixed dimension as input, and applies a set of transposed convolutions, batch normalization, and ReLu activation to it, and generates an image of the required size. Before looking into the generator implementation, let's look at defining transposed convolution and batch normalization.

Transposed convolutions

Transposed convolutions are also called **fractionally strided convolutions**. They work in the opposite way to how convolution works. Intuitively, they try to calculate how the input vector can be mapped to higher dimensions. Let's look at the following figure to understand it better:



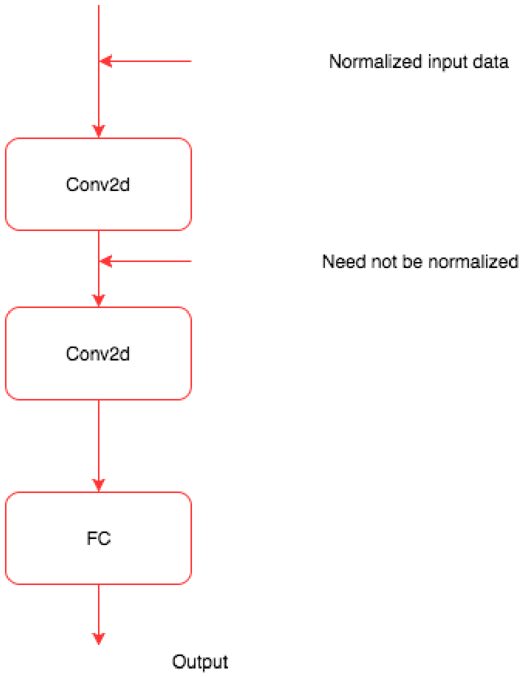
This diagram is referenced from Theano (another popular deep learning framework) documentation (`http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html`). If you want to explore more about how strided convolutions work, I strongly recommend you read this article from the Theano documentation. What is important for us is, that it helps to convert a vector to a tensor of required dimensions, and we can train the values of the kernels by backpropagation.

Batch normalization

We have already observed a couple of times that all the features that are being passed to either machine learning or deep learning algorithms are normalized; that is, the values of the features are centered to zero by subtracting the mean from the data, and giving the data a unit standard deviation by dividing the data by its standard deviation. We would generally do this by using the PyTorch `torchvision.Normalize` method. The following code shows an example:

```
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
```

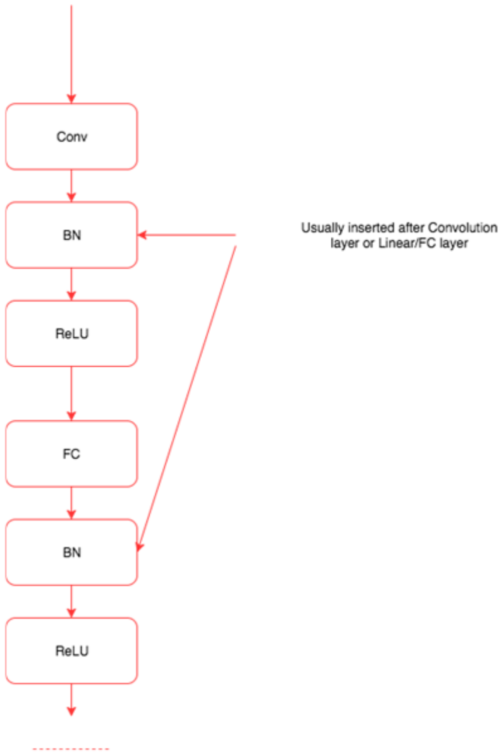
In all the examples we have seen, the data is normalized just before it enters a neural network; there is no guarantee that the intermediate layers get a normalized input. The following figure shows how the intermediate layers in the neural network fail to get normalized data:



Batch normalization acts like an intermediate function, or a layer which normalizes the intermediate data when the mean and variance are changing over time during training. Batch normalization was introduced in 2015 by Ioffe and Szegedy (`https://arxiv.org/abs/1502.03167`). Batch normalization behaves differently during training and validation or testing. During training, the mean and variance is calculated for the data in the batch. For validation and testing, the global values are used. All we need to understand to use it is that it normalizes the intermediate data. Some of the key advantages of using batch normalization are that it:

- Improves gradient flow through the network, thus helping us build deeper networks
- Allows higher learning rates
- Reduces the strong dependency of initialization
- Acts as a form of regularization and reduces the dependency of dropout

Most of the modern architectures, such as ResNet and Inception, extensively use batch normalization in their architectures. Batch normalization layers are introduced after a convolution layer or linear/fully connected layers, as shown in the following image:


Conv
BN
ReLU
FC
BN
ReLU
Usually inserted after Convolution layer or Linear/FC layer

By now, we have an intuitive understanding of the key components of a generator network.

Generator

Let's quickly look at the following generator network code, and then discuss the key features of the generator network:

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d(nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input):
        output = self.main(input)
        return output


netG = Generator()
```

```
netG.apply(weights_init)
print(netG)
```

In most of the code examples we have seen, we use a bunch of different layers and then define the flow in the `forward` method. In the generator network, we define the layers and the flow of the data inside the `__init__` method using a sequential model.

The model takes as input a tensor of size `nz`, and then passes it on to a transposed convolution to map the input to the image size that it needs to generate. The `forward` function passes on the input to the sequential module and returns the output.

The last layer of the generator network is a `tanh` layer, which limits the range of values the network can generate.

Instead of using the same random weights, we initialize the model with weights as defined in the paper. The following is the weight initialization code:

```
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        m.weight.data.normal_(0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        m.weight.data.normal_(1.0, 0.02)
        m.bias.data.fill_(0)
```

We call the `weight` function by passing the function to the generator object, `netG`. Each layer is passed on to the function; if the layer is a convolution layer we initialize the weights differently, and if it is a `BatchNorm`, then we initialize it a bit differently. We call the function on the network object using the following code:

```
netG.apply(weights_init)
```

Defining the discriminator network

Let's quickly look at the following discriminator network code, and then discuss the key features of the discriminator network:

```
class Discriminator(nn.Module):
    def __init__(self):
        super(_netD, self).__init__()
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        output = self.main(input)
        return output.view(-1, 1).squeeze(1)


netD = Discriminator()
netD.apply(weights_init)
print(netD)
```

There are two important things in the previous network, namely, the usage of **leaky ReLU** as an activation function, and the usage of sigmoid as the last activation layer. First, let's understand what Leaky ReLU is.

Leaky ReLU is an attempt to fix the dying ReLU problem. Instead of the function returning zero when the input is negative, leaky ReLU will output a very small number like 0.001. In the paper, it is shown that using leaky ReLU improves the efficiency of the discriminator.

Another important difference is not using fully connected layers at the end of the discriminator. It is common to see the last fully connected layers being replaced by global average pooling. But using global average pooling reduces the rate of the convergence speed (number of iterations to build an accurate classifier). The last convolution layer is flattened and passed to a sigmoid layer.

Other than these two differences, the rest of the network is similar to the other image classifier networks we have seen in the book.

Defining loss and optimizer

We will define a binary cross-entropy loss and two optimizers, one for the generator and another one for the discriminator, in the following code:

```
criterion = nn.BCELoss()

# setup optimizer
optimizerD = optim.Adam(netD.parameters(), lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr, betas=(beta1, 0.999))
```

Up to this point, it is very similar to what we have seen in all our previous examples. Let's explore how we can train the generator and discriminator.

Training the discriminator

The loss of the discriminator network depends on how it performs on real images and how it performs on fake images generated by the generator network. The loss can be defined as:

$$loss = maximize \log(D(x)) + \log(1-D(G(z)))$$

So, we need to train the discriminator with real images and the fake images generated by the generator network.

Training the discriminator with real images

Let's pass some real images as ground truth to train discriminator.

First, we will take a look at the code for doing the same and then explore the important features:

```
output = netD(inputv)
errD_real = criterion(output, labelv)
errD_real.backward()
```

In the previous code, we are calculating the loss and the gradients required for the discriminator image. The `inputv` and `labelv` represent the input image from the `CIFAR10` dataset and labels, which is one for real images. It is pretty straightforward, as it is similar to what we do for other image classifier networks.

Training the discriminator with fake images

Now pass some random images to train discriminator.

Let's look at the code for it and then explore the important features:

```
fake = netG(noisev)
output = netD(fake.detach())
errD_fake = criterion(output, labelv)
errD_fake.backward()
optimizerD.step()
```

The first line in this code passes a vector with a size of 100, and the generator network (`netG`) generates an image. We pass on the image to the discriminator for it to identify whether the image is real or fake. We do not want the generator to get trained, as the discriminator is getting trained. So, we remove the fake image from its graph by calling the `detach` method on its variable. Once all the gradients are calculated, we call the `optimizer` to train the discriminator.

Training the generator network

Let's look at the code for it and then explore the important features:

```
netG.zero_grad()
labelv = Variable(label.fill_(real_label)) # fake labels are real for generat
output = netD(fake)
errG = criterion(output, labelv)
errG.backward()
optimizerG.step()
```

It looks similar to what we did while we trained the discriminator on fake images, except for some key differences. We are passing the same fake images created by the generator, but this time we are not detaching it from the graph that produced it, because we want the generator to be trained. We calculate the loss (`errG`) and calculate the gradients. Then we call the generator optimizer, as we want only the generator to be trained, and we repeat this entire process for several iterations before we have the generator producing slightly realistic images.

Training the complete network

We looked at individual pieces of how a GAN is trained. Let's summarize them as follows and look at the complete code that will be used to train the GAN network we created:

- Train the discriminator network with real images
- Train the discriminator network with fake images
- Optimize the discriminator
- Train the generator based on the discriminator feedback
- Optimize the generator network alone

We will use the following code to train the network:

```
for epoch in range(niter):
    for i, data in enumerate(dataloader, 0):
        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        # train with real
        netD.zero_grad()
        real, _ = data
        batch_size = real.size(0)
        if torch.cuda.is_available():
            real = real.cuda()
        input.resize_as_(real).copy_(real)
        label.resize_(batch_size).fill_(real_label)
        inputv = Variable(input)
        labelv = Variable(label)

        output = netD(inputv)
        errD_real = criterion(output, labelv)
        errD_real.backward()
        D_x = output.data.mean()

        # train with fake
        noise.resize_(batch_size, nz, 1, 1).normal_(0, 1)
        noisev = Variable(noise)
        fake = netG(noisev)
        labelv = Variable(label.fill_(fake_label))
        output = netD(fake.detach())
        errD_fake = criterion(output, labelv)
        errD_fake.backward()
        D_G_z1 = output.data.mean()
        errD = errD_real + errD_fake
        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        labelv = Variable(label.fill_(real_label)) # fake labels are real for
        output = netD(fake)
        errG = criterion(output, labelv)
        errG.backward()
        D_G_z2 = output.data.mean()
        optimizerG.step()

        print('[%d/%d][%d/%d] Loss_D: %.4f Loss_G: %.4f D(x): %.4f D(G(z)): %
              % (epoch, niter, i, len(dataloader),
                 errD.data[0], errG.data[0], D_x, D_G_z1, D_G_z2))
        if i % 100 == 0:
            vutils.save_image(real_cpu,
                    '%s/real_samples.png' % outf,
                    normalize=True)
            fake = netG(fixed_noise)
            vutils.save_image(fake.data,
                    '%s/fake_samples_epoch_%03d.png' % (outf, epoch),
                    normalize=True)
```

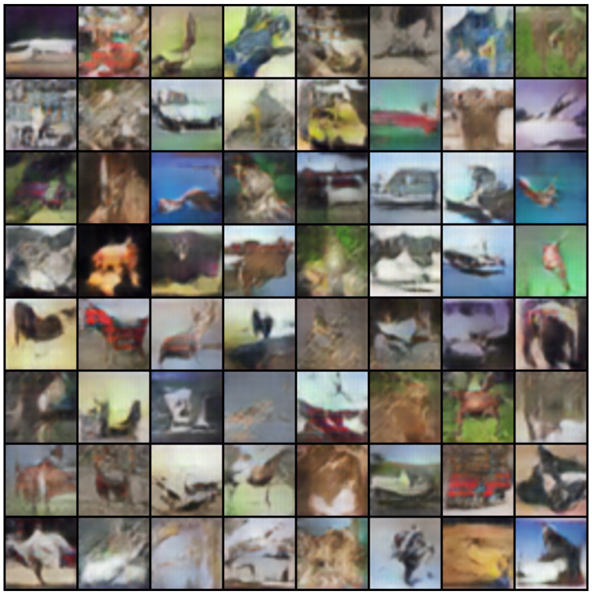
The `vutils.save_image` will take a tensor and save it as an image. If provided with a mini-batch of images, then it saves them as a grid of images.

In the following sections, we will take a look at what the generated images and the real images look like.

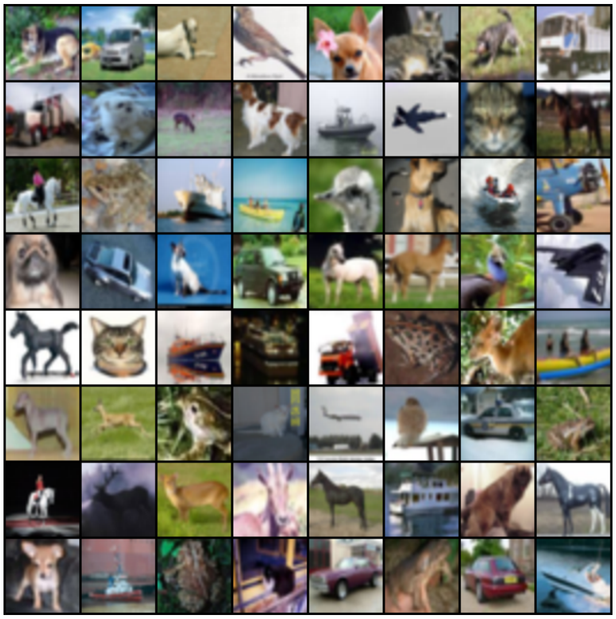
Inspecting the generated images

So, let's compare the generated images and the real images.

The generated images will be as follows:



The real images will be as follows:



Comparing both sets of images, we can see that our GAN was able to learn how to generate images. Apart from training to generate new images, we also have a discriminator, which can be used for classification problems. The discriminator learns important features about the images which can be used for classification tasks when there is a limited amount of labeled data available. When there is limited labeled data, we can train a GAN that will give us a classifier, which can be used to extract features—and a classifier module can be built on top of it.

In the next section, we will train a deep learning algorithm to generate text.

Language modeling

We will learn how to teach a **recurrent neural network** (**RNN**) how it can create a sequence of text. In simple words, the RNN model that we will build now will be able to predict the next word, given some context. This is just like the *Swift* app on your phone, which guesses the next word that you are typing. The ability to generate sequential data has applications in many different areas, such as:

- Image captioning
- Speech recognition
- Language translation
- Automatic email reply

We learnt in `Chapter 6`, *Deep Learning with Sequence Data and Text,* that RNNs are tough to train. So, we will be using a variant of RNN called **Long Short-Term Memory** (**LSTM**). The development of the LSTM algorithm started in 1997 but became popular in the last few years. It became popular due to the availability of powerful hardware and quality data, and some advancements such as dropout also helped in training better LSTM models much more easily than previously.

It is quite popular to use LSTM models to generate either a character-level language model or a word-level language model. In character-level language modeling, we give one character and the

LSTM model is trained to predict the next character, whereas in word-level language modeling, we give a word and the LSTM model predicts the next word. In this section, we will be building a word-level language model using the PyTorch LSTM model. Just like training any other module, we will be following the standard steps:

- Preparing the data
- Generating batches of data
- Defining a model based on LSTM
- Training the model
- Testing the model

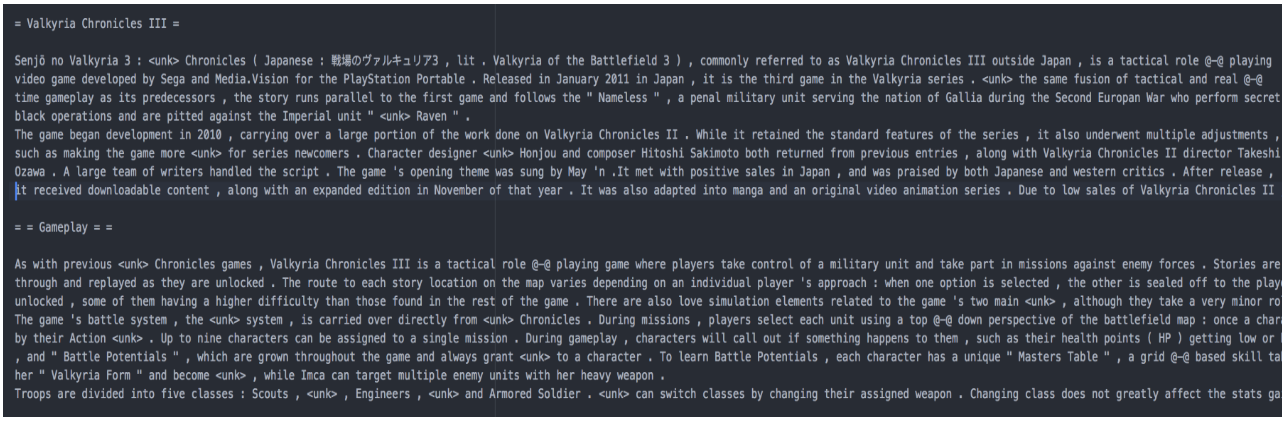
This section is inspired from a slightly simplified version of the word language modeling example available in PyTorch at `https://github.com/pytorch/examples/tree/master/word_language_model`.

Preparing the data

For this example, we use a dataset called `WikiText2`. The `WikiText language modeling` dataset is a collection of over 100 million tokens extracted from the set of verified Good and Featured articles on Wikipedia. Compared to the preprocessed version of **Penn Treebank** (**PTB**), another popularly-used dataset, `WikiText-2` is over two times larger. The `WikiText` dataset also features a far larger vocabulary and retains the original case, punctuation, and numbers. The dataset contains full articles and, as a result, it is well suited for models that take advantage of long term dependency.

The dataset was introduced in a paper called *Pointer Sentinel Mixture Models* (`https://arxiv.org/abs/1609.07843`). The paper talks about solutions that can be used for solving a specific problem, where the LSTM with a softmax layer has difficulty in predicting rare words, though the context is unclear. Let's not worry about this for now, as it is an advanced concept and out of the scope of this book.

The following screenshot shows what the data looks like inside the WikiText dump:



```
= Valkyria Chronicles III =

Senjō no Valkyria 3 : <unk> Chronicles ( Japanese : 戦場のヴァルキュリア3 , lit . Valkyria of the Battlefield 3 ) , commonly referred to as Valkyria Chronicles III outside Japan , is a tactical role @-@ playing
video game developed by Sega and Media.Vision for the PlayStation Portable . Released in January 2011 in Japan , it is the third game in the Valkyria series . <unk> the same fusion of tactical and real @-@
time gameplay as its predecessors , the story runs parallel to the first game and follows the " Nameless " , a penal military unit serving the nation of Gallia during the Second Europan War who perform secret
black operations and are pitted against the Imperial unit " <unk> Raven " .
The game began development in 2010 , carrying over a large portion of the work done on Valkyria Chronicles II . While it retained the standard features of the series , it also underwent multiple adjustments ,
such as making the game more <unk> for series newcomers . Character designer <unk> Honjou and composer Hitoshi Sakimoto both returned from previous entries , along with Valkyria Chronicles II director Takeshi
Ozawa . A large team of writers handled the script . The game 's opening theme was sung by May 'n .It met with positive sales in Japan , and was praised by both Japanese and western critics . After release ,
it received downloadable content , along with an expanded edition in November of that year . It was also adapted into manga and an original video animation series . Due to low sales of Valkyria Chronicles II

= = Gameplay = =

As with previous <unk> Chronicles games , Valkyria Chronicles III is a tactical role @-@ playing game where players take control of a military unit and take part in missions against enemy forces . Stories are
through and replayed as they are unlocked . The route to each story location on the map varies depending on an individual player 's approach : when one option is selected , the other is sealed off to the playe
unlocked , some of them having a higher difficulty than those found in the rest of the game . There are also love simulation elements related to the game 's two main <unk> , although they take a very minor ro
The game 's battle system , the <unk> system , is carried over directly from <unk> Chronicles . During missions , players select each unit using a top @-@ down perspective of the battlefield map : once a chara
by their Action <unk> . Up to nine characters can be assigned to a single mission . During gameplay , characters will call out if something happens to them , such as their health points ( HP ) getting low or
, and " Battle Potentials " , which are grown throughout the game and always grant <unk> to a character . To learn Battle Potentials , each character has a unique " Masters Table " , a grid @-@ based skill ta
her " Valkyria Form " and become <unk> , while Imca can target multiple enemy units with her heavy weapon .
Troops are divided into five classes : Scouts , <unk> , Engineers , <unk> and Armored Soldier . <unk> can switch classes by changing their assigned weapon . Changing class does not greatly affect the stats ga
```

As usual, `torchtext` makes it easier to use the dataset, by providing abstractions over downloading and reading the dataset. Let's look at the code that does that:

```
TEXT = d.Field(lower=True, batch_first=True)
train, valid, test = datasets.WikiText2.splits(TEXT,root='data')
```

The previous code takes care of downloading the `WikiText2` data and splits it into `train`, `valid`, and `test` datasets. The key difference in language modeling is how the data is processed. All the text data that we had in `WikiText2` is stored in one long tensor. Let's look at the following code and the results, to understand how the data is processed better:

```
print(len(train[0].text))

#output
2088628
```

As we can see from the previous results, we have only one example field and it contains all the text. Let's also quickly look at how the text is represented:

```
print(train[0].text[:100])

#Results of first 100 tokens

'<eos>', '=', 'valkyria', 'chronicles', 'iii', '=', '<eos>', '<eos>', 'senjō'
```

Now, take a quick look at the image that showed the initial text and how it is being tokenized. Now we have a long sequence, of length `2088628`, representing `WikiText2`. The next important thing is how we batch the data.

Generating the batches

Let's take a look at the code and understand the two key things involved in the batching of sequential data:

```
train_iter, valid_iter, test_iter = data.BPTTIterator.splits(
    (train, valid, test), batch_size=20, bptt_len=35, device=0)
```

There are two important things that are going through this method. One is `batch_size`, and another is `bptt_len`, called **backpropagation through time**. It gives a brief idea about how data is transformed through each phase.

Batches

Processing the entire data as a sequence is quite challenging and not computationally efficient. So, we break the sequence data into multiple batches, and treat each as a separate sequence. Though it may not sound straightforward, it works a lot better, as the model can learn quicker from batches of data. Let's take the example where the English alphabet is sequenced and we split it into batches.

Sequence: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z.

When we convert the preceding alphabet sequence into four batches, we get:

a g m s y

b h n t z

c i o u

d j p v

e k q w

f l r x

In most of the cases, we would end up trimming the last extra words or tokens that form a small batch, since it doesn't have a great effect on text modeling.

For the example `WikiText2`, when we split the data into 20 batches, we would get each batch with elements 104431.

Backpropagation through time

The other important variable that we see go through the iterator is **backpropagation through time** (**BPTT**). What it actually means is, the sequence length the model needs to remember. The higher the number, the better—but the complexity of the model and the GPU memory required for the model also increase.

To understand it better, let's look at how we can split the previous batched alphabet data into sequences of length two:

a g m s

b h n t

The previous example will be passed to the model as input, and the output will be from the sequence but containing the next values:

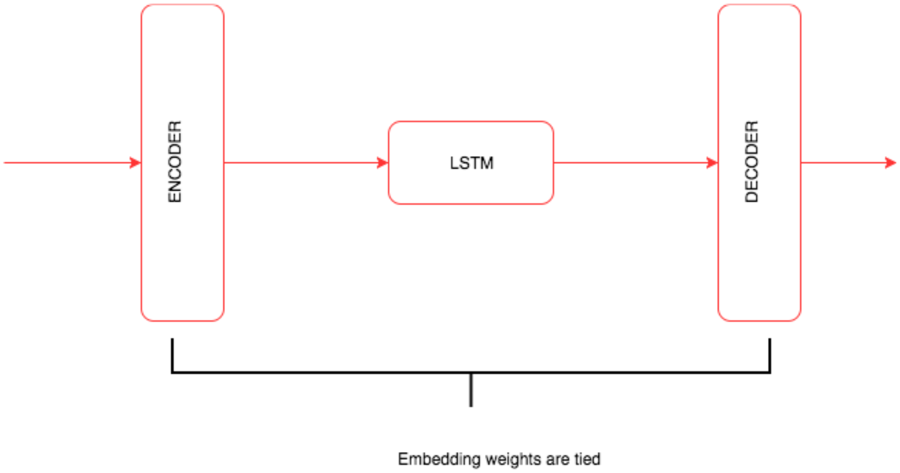
b h n t

c I o u

For the example `WikiText2`, when we split the batched data, we get data of size 30, 20 for each batch where *30* is the sequence length.

Defining a model based on LSTM

We defined a model that is a bit similar to the networks that we saw in `Chapter 6`, *Deep Learning with Sequence Data and Text,* but it has some key differences. The high-level architecture of the network looks like the following image:



As usual, let's take a look at the code and then walk through the key parts of it:

```
class RNNModel(nn.Module):
    def __init__(self,ntoken,ninp,nhid,nlayers,dropout=0.5,tie_weights=False)
        #ntoken represents the number of words in vocabulary.
        #ninp Embedding dimension for each word ,which is the input for the I
        #nlayer Number of layers required to be used in the LSTM .
        #Dropout to avoid overfitting.
        #tie_weights - use the same weights for both encoder and decoder.
        super().__init__()
        self.drop = nn.Dropout()
        self.encoder = nn.Embedding(ntoken,ninp)
        self.rnn = nn.LSTM(ninp,nhid,nlayers,dropout=dropout)
        self.decoder = nn.Linear(nhid,ntoken)
        if tie_weights:
            self.decoder.weight = self.encoder.weight

        self.init_weights()
        self.nhid = nhid
        self.nlayers = nlayers

    def init_weights(self):
        initrange = 0.1
        self.encoder.weight.data.uniform_(-initrange,initrange)
        self.decoder.bias.data.fill_(0)
        self.decoder.weight.data.uniform_(-initrange,initrange)

    def forward(self,input,hidden):
        emb = self.drop(self.encoder(input))
        output,hidden = self.rnn(emb,hidden)
        output = self.drop(output)
        s = output.size()
        decoded = self.decoder(output.view(s[0]*s[1],s[2]))
        return decoded.view(s[0],s[1],decoded.size(1)),hidden

    def init_hidden(self,bsz):
        weight = next(self.parameters()).data

        return (Variable(weight.new(self.nlayers,bsz,self.nhid).zero_()),Vari
```

In the `__init__` method, we create all the layers such as embedding, dropout, RNN, and decoder. In earlier language models, embeddings were not generally used in the last layer. The use of embeddings, and tying the initial embedding along with the embeddings of the final output layer, improves the accuracy of the language model. This concept was introduced in the papers *Using the Output Embedding to Improve Language Models* (`https://arxiv.org/abs/1608.05859`) by Press and Wolf in 2016, and *Tying Word Vectors and Word Classifiers: A Loss Framework for Language Modeling* (`https://arxiv.org/abs/1611.01462`) by Inan and his co-authors in 2016. Once we have made the weights of encoder and decoder tied, we call the `init_weights` method to initialize the weights of the layer.

The `forward` function stitches all the layers together. The last linear layers map all the output activations from the LSTM layer to the embeddings that are of the size of the vocabulary. The flow of the `forward` function input is passed through the embedding layer and then passed on to an RNN (in this case, an LSTM), and then to the decoder, another linear layer.

Defining the train and evaluate functions

The training of the model is very similar to what we saw in all the previous examples in this book. There are a few important changes that we need to make so that the trained model works better. Let's look at the code and its key parts:

```
criterion = nn.CrossEntropyLoss()

def trainf():
    # Turn on training mode which enables dropout.
    lstm.train()
    total_loss = 0
    start_time = time.time()
    hidden = lstm.init_hidden(batch_size)
    for i,batch in enumerate(train_iter):
        data, targets = batch.text,batch.target.view(-1)
        # Starting each batch, we detach the hidden state from how it was pre
        # If we didn't, the model would try backpropagating all the way to st
        hidden = repackage_hidden(hidden)
        lstm.zero_grad()
        output, hidden = lstm(data, hidden)
        loss = criterion(output.view(-1, ntokens), targets)
        loss.backward()

        # `clip_grad_norm` helps prevent the exploding gradient problem in RN
        torch.nn.utils.clip_grad_norm(lstm.parameters(), clip)
        for p in lstm.parameters():
            p.data.add_(-lr, p.grad.data)

        total_loss += loss.data

        if i % log_interval == 0 and i > 0:
            cur_loss = total_loss[0] / log_interval
            elapsed = time.time() - start_time
            (print('| epoch {:3d} | {:5d}/{:5d} batches | lr {:02.2f} | ms/ba
            total_loss = 0
            start_time = time.time()
```

Since we are using dropout in our model, we need to use it differently during training and for validation/test datasets. Calling `train()` on the model will ensure dropout is active during training, and calling `eval()` on the model will ensure that dropout is used differently:

```
lstm.train()
```

For an LSTM model, along with the input, we also need to pass the hidden variables. The `init_hidden` function will take the batch size as input and then return a hidden variable, which can be used along with the inputs. We can iterate through the training data and pass the input data to the model. Since we are processing sequence data, starting with a new hidden state (randomly initialized) for every iteration will not make sense. So, we will use the hidden state from the previous iteration after removing it from the graph by calling the `detach` method. If we do not call the `detach` method, then we end up calculating gradients for a very long sequence until we run out of GPU memory.

We then pass on the input to the LSTM model and calculate loss using `CrossEntropyLoss`. Using the previous values of the hidden state is implemented in the following `repackage_hidden` function:

```
def repackage_hidden(h):
    """Wraps hidden states in new Variables, to detach them from their histor
    if type(h) == Variable:
        return Variable(h.data)
    else:
        return tuple(repackage_hidden(v) for v in h)
```

RNNs and their variants, such as LSTM and the **Gated Recurrent Unit** (**GRU**), suffer from a problem called **exploding gradients**. One simple trick to avoid the problem is to clip the gradients, which is done in the following code:

```
torch.nn.utils.clip_grad_norm(lstm.parameters(), clip)
```

We manually adjust the values of the parameters by using the following code. Implementing an optimizer manually gives more flexibility than using a prebuilt optimizer:

```
  for p in lstm.parameters():
      p.data.add_(-lr, p.grad.data)
```

We are iterating through all the parameters and adding up the value of the gradients, multiplied by the learning rate. Once we update all the parameters, we log all the statistics such as time, loss, and perplexity.

We write a similar function for validation, where we call the `eval` method on the model. The `evaluate` function is defined using the following code:

```
def evaluate(data_source):
    # Turn on evaluation mode which disables dropout.
    lstm.eval()
    total_loss = 0
    hidden = lstm.init_hidden(batch_size)
    for batch in data_source:
        data, targets = batch.text,batch.target.view(-1)
        output, hidden = lstm(data, hidden)
        output_flat = output.view(-1, ntokens)
        total_loss += len(data) * criterion(output_flat, targets).data
        hidden = repackage_hidden(hidden)
    return total_loss[0]/(len(data_source.dataset[0].text)//batch_size)
```

Most of the training logic and evaluation logic is similar, except for calling `eval` and not updating the parameters of the model.

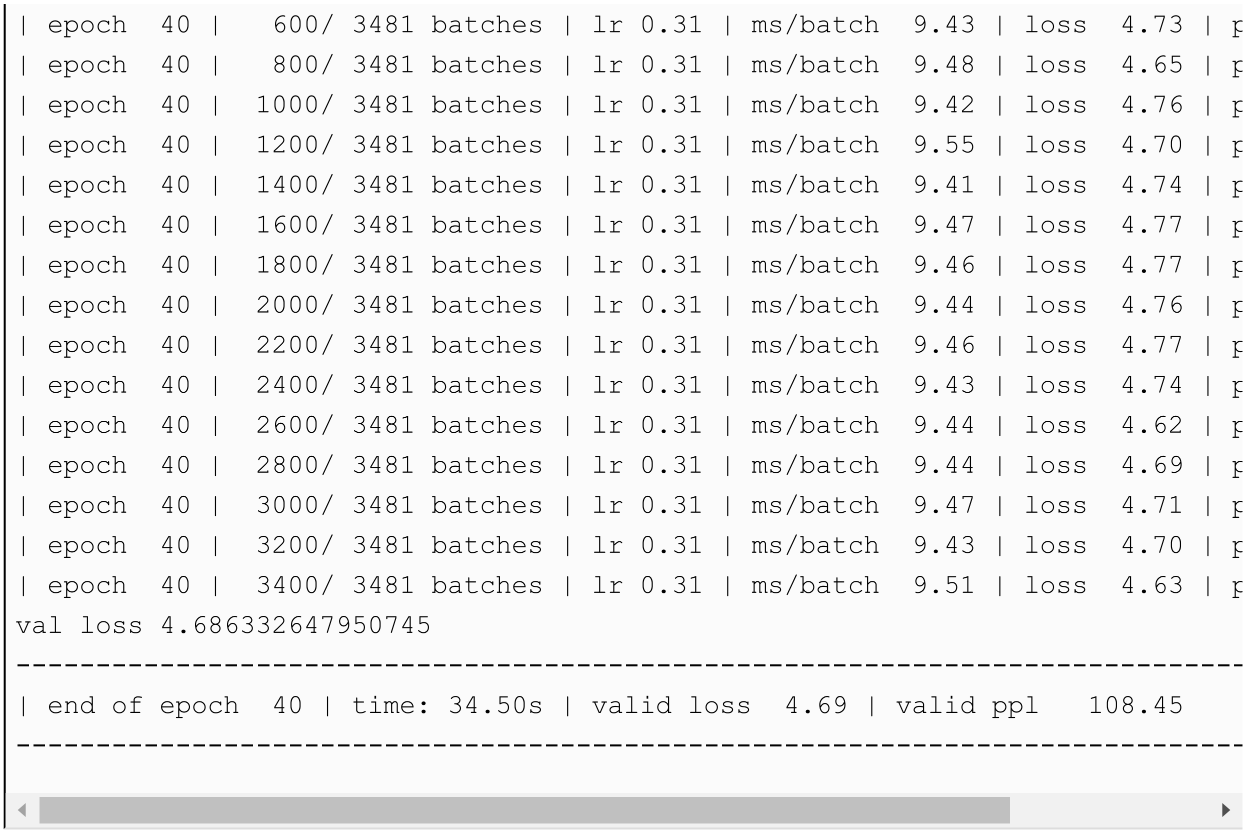
Training the model

We train the model for multiple epochs and validate it using the following code:

```
# Loop over epochs.
best_val_loss = None
epochs = 40

for epoch in range(1, epochs+1):
    epoch_start_time = time.time()
    trainf()
    val_loss = evaluate(valid_iter)
    print('-' * 89)
    print('| end of epoch {:3d} | time: {:5.2f}s | valid loss {:5.2f} | '
        'valid ppl {:8.2f}'.format(epoch, (time.time() - epoch_start_time),
                                   val_loss, math.exp(val_loss)))
    print('-' * 89)
    if not best_val_loss or val_loss < best_val_loss:
        best_val_loss = val_loss
    else:
        # Anneal the learning rate if no improvement has been seen in the val
        lr /= 4.0
```

The previous code is training the model for `40` epochs, and we start with a high-learning rate of 20 and reduce it further when the validation loss saturates. Running the model for 40 epochs gives a `ppl` score of approximately `108.45`. The following code block contains the logs when the model was last run:

```
-----------------------------------------------------------------------------
| end of epoch  39 | time: 34.16s | valid loss  4.70 | valid ppl   110.01
-----------------------------------------------------------------------------
| epoch  40 |   200/ 3481 batches | lr 0.31 | ms/batch 11.47 | loss  4.77 | p
| epoch  40 |   400/ 3481 batches | lr 0.31 | ms/batch  9.56 | loss  4.81 | p
```



```
| epoch  40 |   600/ 3481 batches | lr 0.31 | ms/batch  9.43 | loss  4.73 | p
| epoch  40 |   800/ 3481 batches | lr 0.31 | ms/batch  9.48 | loss  4.65 | p
| epoch  40 |  1000/ 3481 batches | lr 0.31 | ms/batch  9.42 | loss  4.76 | p
| epoch  40 |  1200/ 3481 batches | lr 0.31 | ms/batch  9.55 | loss  4.70 | p
| epoch  40 |  1400/ 3481 batches | lr 0.31 | ms/batch  9.41 | loss  4.74 | p
| epoch  40 |  1600/ 3481 batches | lr 0.31 | ms/batch  9.47 | loss  4.77 | p
| epoch  40 |  1800/ 3481 batches | lr 0.31 | ms/batch  9.46 | loss  4.77 | p
| epoch  40 |  2000/ 3481 batches | lr 0.31 | ms/batch  9.44 | loss  4.76 | p
| epoch  40 |  2200/ 3481 batches | lr 0.31 | ms/batch  9.46 | loss  4.77 | p
| epoch  40 |  2400/ 3481 batches | lr 0.31 | ms/batch  9.43 | loss  4.74 | p
| epoch  40 |  2600/ 3481 batches | lr 0.31 | ms/batch  9.44 | loss  4.62 | p
| epoch  40 |  2800/ 3481 batches | lr 0.31 | ms/batch  9.44 | loss  4.69 | p
| epoch  40 |  3000/ 3481 batches | lr 0.31 | ms/batch  9.47 | loss  4.71 | p
| epoch  40 |  3200/ 3481 batches | lr 0.31 | ms/batch  9.43 | loss  4.70 | p
| epoch  40 |  3400/ 3481 batches | lr 0.31 | ms/batch  9.51 | loss  4.63 | p
val loss 4.686332647950745
-----------------------------------------------------------------------------
| end of epoch  40 | time: 34.50s | valid loss  4.69 | valid ppl   108.45
-----------------------------------------------------------------------------
```

In the last few months, researchers started exploring the previous approach to create a language model for creating pretrained embeddings. If you are more interested in this approach, I would strongly recommend you read the paper *Fine-tuned Language Models for Text Classification* (`https://arxiv.org/abs/1801.06146`) by Jeremy Howard and Sebastian Ruder, where they go into a lot of detail on how language modeling techniques can be used to prepare domain-specific word embeddings, which can later be used for different NLP tasks, such as text classification problems.

Summary

In this chapter, we covered how to train deep learning algorithms that can generate artistic style transfers using generative networks, new images using GAN and DCGAN, and generate text using LSTM networks.

In the next chapter, we will cover some of the modern architectures, such as ResNet and Inception, for building better computer vision models and models such as sequence-to-sequence, which can be used for building language translation and image captioning.

Modern Network Architectures

In the last chapter, we explored how deep learning algorithms can be used to create artistic images, create new images based on existing datasets, and generate text. In this chapter, we will introduce you to different network architectures that power modern computer vision applications and natural language systems. Some of the architectures that we will look at in this chapter are:

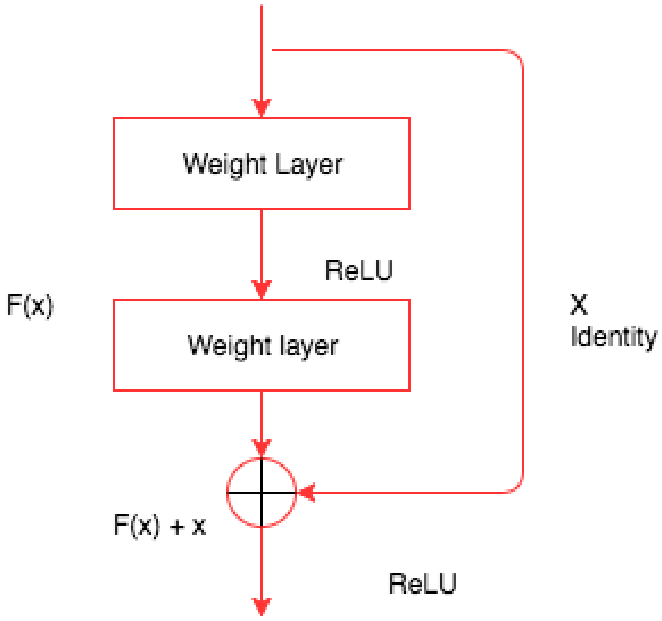
- ResNet
- Inception
- DenseNet
- Encoder-decoder architecture

Modern network architectures

One of the important things that we do when the deep learning model fails to learn is to add more layers to the model. As you add layers the model accuracy improves and then starts saturating. It starts degrading as you keep on adding more layers. Adding more layers beyond a certain number will add certain challenges, such as vanishing or exploding gradients, which is partially solved by carefully initializing weights and introducing intermediate normalizing layers. Modern architectures, such as **residual network** (**ResNet**) and Inception, try to solve this problem by introducing different techniques, such as residual connections.

ResNet

ResNet solves these problems by explicitly letting the layers in the network fit a residual mapping by adding a shortcut connection. The following image shows how ResNet works:



In all the networks we have seen, we try to find a function that maps the input (x) to its output ($H(x)$) by stacking different layers. But the authors of ResNet proposed a fix; instead of trying to learn an underlying mapping from x to $H(x)$, we learn the difference between the two, or the residual. Then, to calculate $H(x)$, we can just add the residual to the input. Say the residual is $F(x) = H(x) - x$; instead of trying to learn $H(x)$ directly, we try to learn $F(x) + x$.

Each ResNet block consists of a series of layers, and a shortcut connection adding the input of the block to the output of the block. The add operation is performed element-wise and the inputs and outputs need to be of the same size. If they are of a different size, then we can use paddings. The following code demonstrates what a simple ResNet block would look like:

```
class ResNetBasicBlock(nn.Module):

    def __init__(self,in_channels,out_channels,stride):

        super().__init__()
        self.conv1 = nn.Conv2d(in_channels,out_channels,kernel_size=3,stride=
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels,out_channels,kernel_size=3,stride
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.stride = stride

    def forward(self,x):

        residual = x
        out = self.conv1(x)
        out = F.relu(self.bn1(out),inplace=True)
        out = self.conv2(out)
        out = self.bn2(out)
        out += residual
        return F.relu(out)
```

The `ResNetBasicBlock` contains an `init` method which initializes all the different layers, such as the convolution layer, batch normalization, and ReLU layers. The `forward` method is almost similar to what we have seen up until now, except that the input is being added back to the layer's output just before it is returned.

The PyTorch `torchvision` package provides an out-of-the-box ResNet model with different layers. Some of the different models available are:

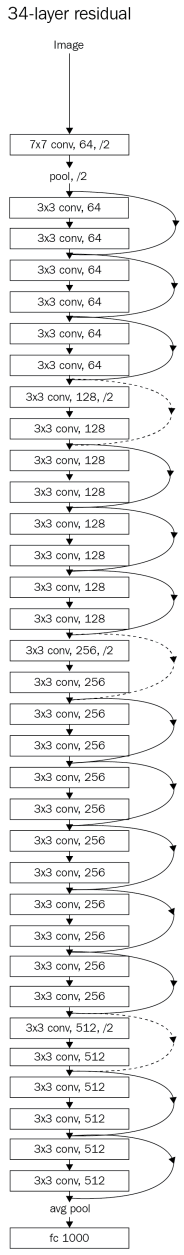
- ResNet-18
- ResNet-34

- ResNet-50
- ResNet-101
- ResNet-152

We can also use any of these models for transfer learning. The `torchvision` instance enables us to simply create one of these models and use them. We have done this a couple of times in the book, and the following code is a refresher for that:

```
from torchvision.models import resnet18

resnet = resnet18(pretrained=False)
```

The following figure shows what a 34-layer ResNet model would look like:


34-layer residual
Image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
avg pool
fc 1000

34 layer ResNet model

We can see how this network consists of multiple ResNet blocks. There have been experiments where teams have tried models as deep as 1,000 layers. For most real-world use cases, my personal recommendation would be to start with a smaller network. Another key advantage of these modern networks is that they need very few parameters compared to models such as VGG, as they avoid using fully connected layers that need lots of parameters to train. Another popular architecture that is being used to solve problems in the computer vision field is **Inception**. Before moving on to Inception architecture, let's train a ResNet model on the `Dogs vs. Cats` dataset. We will use the data that we used in `Chapter 5`*, Deep Learning for Computer Vision,* and will quickly train a model based on features calculated from ResNet. As usual, we will follow these steps to train the model:

- Creating PyTorch datasets
- Creating loaders for training and validation
- Creating the ResNet model
- Extract convolutional features
- Creating a custom PyTorch dataset class for the pre-convoluted features and loader
- Creating a simple linear model
- Training and validating the model

Once done, we are going to repeat this step for Inception and DenseNet. At the end, we will also explore the ensembling technique, where we combine these powerful models to build a new model.

Creating PyTorch datasets

We create a transformation object containing all the basic transformations required and use the `ImageFolder` to load the images from the data directory that we created in `Chapter 5`, *Deep Learning for Computer Vision*. In the following code, we create the datasets:

```
data_transform = transforms.Compose([
        transforms.Resize((299,299)),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])

# For Dogs & Cats dataset
train_dset = ImageFolder('../../chapter5/dogsandcats/train/',transform=data_t
val_dset = ImageFolder('../../chapter5/dogsandcats/valid/',transform=data_tra
classes=2
```

By now, most of the preceding code will be self-explanatory.

Creating loaders for training and validation

We use PyTorch loaders to load the data provided by the dataset in the form of batches, along with all the advantages, such as shuffling the data and using multi-threads, to speed up the process. The following code demonstrates this:

```
train_loader = DataLoader(train_dset,batch_size=32,shuffle=False,num_workers=
val_loader = DataLoader(val_dset,batch_size=32,shuffle=False,num_workers=3)
```

We need to maintain the exact sequence of the data while calculating the pre-convoluted features. When we allow the data to be shuffled, we will not be able to maintain the labels. So, ensure the `shuffle` is `False`, otherwise the required logic needs to be handled inside the code.

Creating a ResNet model

Using the layers of the `resnet34` pretrained model, we create a PyTorch sequential model by discarding the last linear layer. We will use this trained model for extracting features from our images. The following code demonstrates this:

```
#Create ResNet model
my_resnet = resnet34(pretrained=True)

if is_cuda:
    my_resnet = my_resnet.cuda()

my_resnet = nn.Sequential(*list(my_resnet.children())[:-1])

for p in my_resnet.parameters():
    p.requires_grad = False
```

In the preceding code, we created a `resnet34` model available in `torchvision` models. In the following line, we pick all the ResNet layers, excluding the last layer, and create a new model using `nn.Sequential`:

```
for p in my_resnet.parameters():
    p.requires_grad = False
```

The `nn.Sequential` instance allows us to quickly create a model using a bunch of PyTorch layers. Once the model is created, do not forget to set the `requires_grad` parameter to `False`, as this will allow PyTorch not to maintain any space for holding gradients.

Extracting convolutional features

We pass the data from the train and validation data loaders through the model and store the results of the model in a list for further computation. By calculating the pre-convoluted features, we can save a lot of time in training the model, as we will not be calculating these features in every iteration. In the following code, we calculate the pre-convulted features:

```
#For training data

# Stores the labels of the train data
trn_labels = []

# Stores the pre convoluted features of the train data
trn_features = []

#Iterate through the train data and store the calculated features and the lab
for d,la in train_loader:
    o = m(Variable(d.cuda()))
    o = o.view(o.size(0),-1)
    trn_labels.extend(la)
    trn_features.extend(o.cpu().data)

#For validation data

#Iterate through the validation data and store the calculated features and th
val_labels = []
val_features = []
for d,la in val_loader:
    o = m(Variable(d.cuda()))
    o = o.view(o.size(0),-1)
    val_labels.extend(la)
    val_features.extend(o.cpu().data)
```

Once we calculate the pre-convoluted features, we need to create a custom dataset that can pick data from our pre-convoluted features. Let's create a custom dataset and loader for the pre-convoluted features.

Creating a custom PyTorch dataset class for the pre-convoluted features and loader

We have already seen how to create a PyTorch dataset. It should be a subclass of the `torch.utils.data` dataset class and should implement the `__getitem__(self, index)` and `__len__(self)` methods, which return the length of the data in the dataset. In the following code, we implement a custom dataset for the pre-convoluted features:

```
class FeaturesDataset(Dataset):

    def __init__(self,featlst,labellst):
        self.featlst = featlst
        self.labellst = labellst

    def __getitem__(self,index):
        return (self.featlst[index],self.labellst[index])

    def __len__(self):
        return len(self.labellst)
```

Once the custom dataset class is created, creating a data loader for the pre-convoluted features is straightforward, as shown in the following code:

```
#Creating dataset for train and validation
trn_feat_dset = FeaturesDataset(trn_features,trn_labels)
val_feat_dset = FeaturesDataset(val_features,val_labels)
```

```
#Creating data loader for train and validation
trn_feat_loader = DataLoader(trn_feat_dset,batch_size=64,shuffle=True)
val_feat_loader = DataLoader(val_feat_dset,batch_size=64)
```

Now we need to create a simple linear model that can map the pre-convoluted features to the corresponding categories.

Creating a simple linear model

We will create a simple linear model that will map the pre-convoluted features to the respective categories. In this case, the number of categories is two:

```
class FullyConnectedModel(nn.Module):

    def __init__(self,in_size,out_size):
        super().__init__()
        self.fc = nn.Linear(in_size,out_size)

    def forward(self,inp):
        out = self.fc(inp)
        return out

fc_in_size = 8192

fc = FullyConnectedModel(fc_in_size,classes)
if is_cuda:
    fc = fc.cuda()
```

Now, we are good to train our new model and validate the dataset.

Training and validating the model

We will use the same `fit` function that we have been using from `Chapter 5`, *Deep Learning for Computer Vision*. I am not including that here, to save space. The following code snippet contains functionality to train the model and shows the results:

```
train_losses , train_accuracy = [],[]
val_losses , val_accuracy = [],[]
for epoch in range(1,10):
    epoch_loss, epoch_accuracy = fit(epoch,fc,trn_feat_loader,phase='training
    val_epoch_loss , val_epoch_accuracy = fit(epoch,fc,val_feat_loader,phase=
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    val_losses.append(val_epoch_loss)
    val_accuracy.append(val_epoch_accuracy)
```

The result of the preceding code is as follows:

```
#Results
training loss is 0.082 and training accuracy is 22473/23000     97.71
validation loss is   0.1 and validation accuracy is 1934/2000      96.7
training loss is  0.08 and training accuracy is 22456/23000     97.63
validation loss is  0.12 and validation accuracy is 1917/2000     95.85
training loss is 0.077 and training accuracy is 22507/23000     97.86
validation loss is   0.1 and validation accuracy is 1930/2000      96.5
training loss is 0.075 and training accuracy is 22518/23000      97.9
validation loss is 0.096 and validation accuracy is 1938/2000      96.9
training loss is 0.073 and training accuracy is 22539/23000      98.0
validation loss is   0.1 and validation accuracy is 1936/2000      96.8
training loss is 0.073 and training accuracy is 22542/23000     98.01
validation loss is 0.089 and validation accuracy is 1942/2000      97.1
training loss is 0.071 and training accuracy is 22545/23000     98.02
```

```
validation loss is  0.09 and validation accuracy is 1941/2000     97.05
training loss is 0.068 and training accuracy is 22591/23000     98.22
validation loss is 0.092 and validation accuracy is 1934/2000      96.7
training loss is 0.067 and training accuracy is 22573/23000     98.14
validation loss is 0.085 and validation accuracy is 1942/2000      97.1
```

As we can see from the results, the model achieves a 98% training accuracy and 97% validation accuracy. Let's understand another modern architecture and how to use it for calculating pre-convoluted features and use them to train a model.

Inception

In most of the deep learning algorithms we have seen for computer vision models, we either pick up a convolution layer with a filter size of 1 x 1, 3 x 3, 5 x 5, 7 x 7, or a map pooling layer. The Inception module combines convolutions of different filter sizes and concatenates all the outputs together. The following image makes the Inception model clearer:

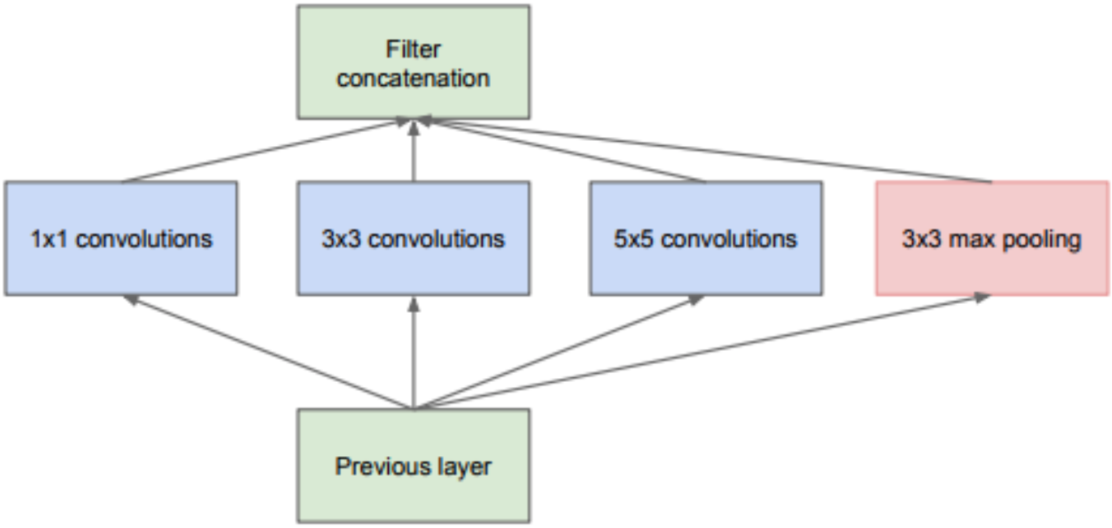

Filter concatenation
1x1 convolutions
3x3 convolutions
5x5 convolutions
3x3 max pooling
Previous layer

Image source: https://arxiv.org/pdf/1409.4842.pdf

In this Inception block image, the convolution of different sizes is applied to the input, and the outputs of all these layers are concatenated. This is the simplest version of an Inception module. There is another variant of an Inception block where we pass the input through a 1 x 1 convolution before passing it through 3 x 3 and 5 x 5 convolutions. A 1 x 1 convolution is used for dimensionality reduction. It helps in solving computational bottlenecks. A 1 x 1 convolution looks at one value at a time and across the channels. For example, using a 10 x 1 x 1 filter on an input size of 100 x 64 x 64 would result in 10 x 64 x 64. The following figure shows the Inception block with dimensionality reductions:

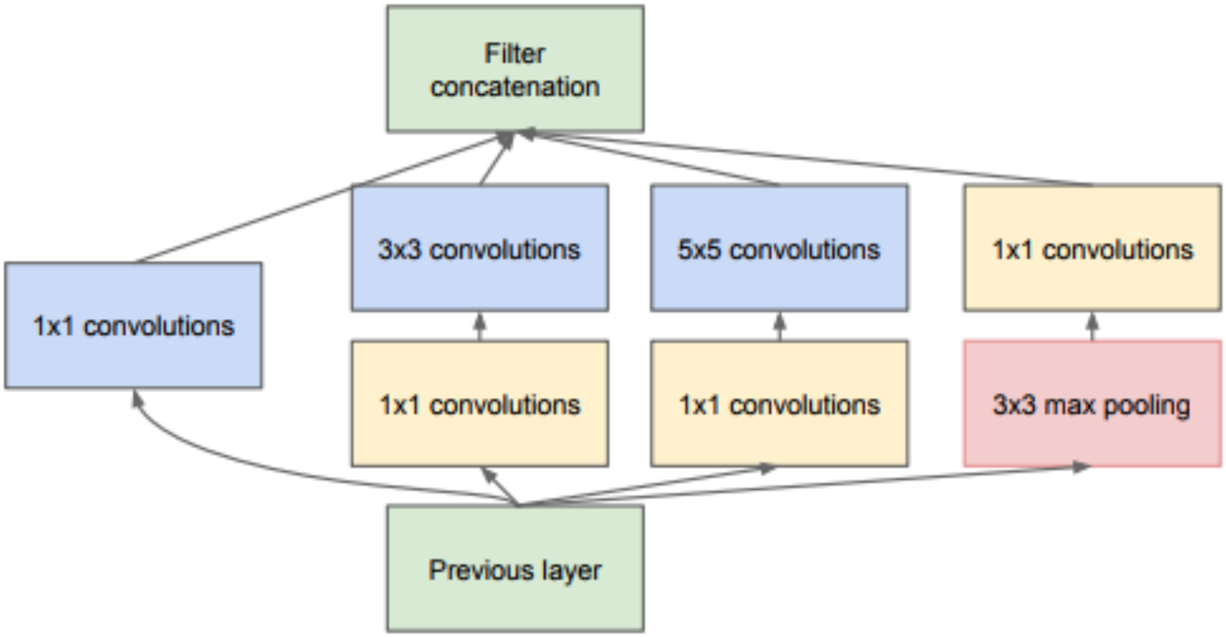

Filter concatenation
1x1 convolutions
3x3 convolutions
5x5 convolutions
1x1 convolutions
1x1 convolutions
1x1 convolutions
3x3 max pooling
Previous layer

Image source: https://arxiv.org/pdf/1409.4842.pdf

Now, let's look at a PyTorch example of what the preceding Inception block would look like:

```
class BasicConv2d(nn.Module):

    def __init__(self, in_channels, out_channels, **kwargs):
        super(BasicConv2d, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, bias=False, **kwargs
        self.bn = nn.BatchNorm2d(out_channels)

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        return F.relu(x, inplace=True)


class InceptionBasicBlock(nn.Module):

    def __init__(self, in_channels, pool_features):
        super().__init__()
        self.branch1x1 = BasicConv2d(in_channels, 64, kernel_size=1)

        self.branch5x5_1 = BasicConv2d(in_channels, 48, kernel_size=1)
        self.branch5x5_2 = BasicConv2d(48, 64, kernel_size=5, padding=2)

        self.branch3x3dbl_1 = BasicConv2d(in_channels, 64, kernel_size=1)
        self.branch3x3dbl_2 = BasicConv2d(64, 96, kernel_size=3, padding=1)

        self.branch_pool = BasicConv2d(in_channels, pool_features, kernel_siz

    def forward(self, x):
        branch1x1 = self.branch1x1(x)

        branch5x5 = self.branch5x5_1(x)
        branch5x5 = self.branch5x5_2(branch5x5)

        branch3x3dbl = self.branch3x3dbl_1(x)
        branch3x3dbl = self.branch3x3dbl_2(branch3x3dbl)

        branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
        branch_pool = self.branch_pool(branch_pool)

        outputs = [branch1x1, branch5x5, branch3x3dbl, branch_pool]
        return torch.cat(outputs, 1)
```

The preceding code contains two classes, `BasicConv2d` and `InceptionBasicBlock`. `BasicConv2d` acts like a custom layer which applies a two-dimensional convolution layer, batch normalization, and a ReLU layer to the input that is passed through. It is good practice to create a new layer when we have a repeating code structure, to make the code look elegant.

The `InceptionBasicBlock` implements what we have in the second Inception figure. Let's go through each smaller snippet and try to understand how it is implemented:

```
branch1x1 = self.branch1x1(x)
```

The preceding code transforms the input by applying a 1 x 1 convolution block:

```
branch5x5 = self.branch5x5_1(x)
branch5x5 = self.branch5x5_2(branch5x5)
```

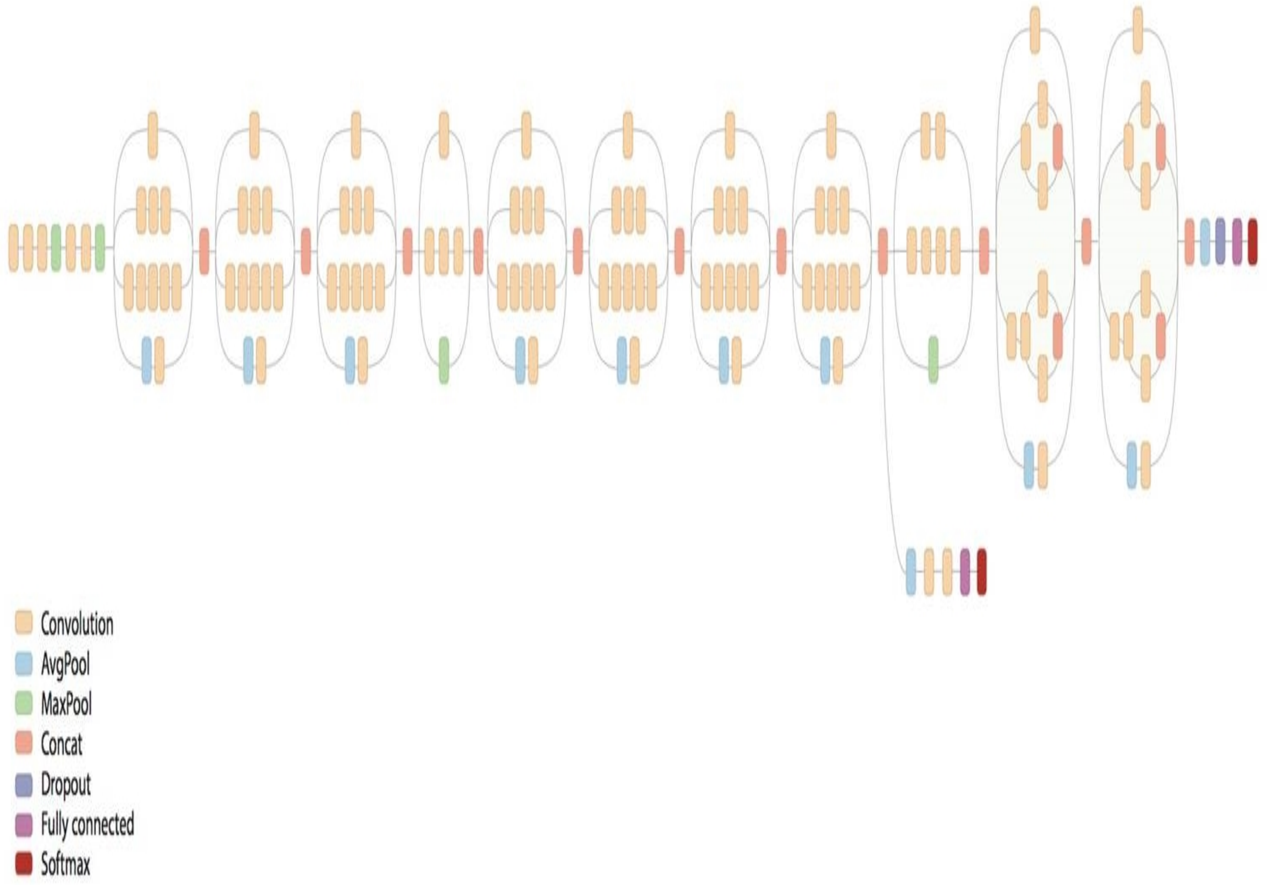
In the preceding code, we transform the input by applying a 1 x 1 convolution block followed by a 5 x 5 convolution block:

```
branch3x3dbl = self.branch3x3dbl_1(x)
branch3x3dbl = self.branch3x3dbl_2(branch3x3dbl)
```

In the preceding code, we transform the input by applying a 1 x 1 convolution block followed by a 3 x 3 convolution block:

```
branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
branch_pool = self.branch_pool(branch_pool)
```

In the preceding code, we apply an average pool along with a 1 x 1 convolution block, and at the end, we concatenate all the results together. An Inception network would consist of several Inception blocks. The following image shows what an Inception architecture would look like:



Inception architecture

The `torchvision` package has an Inception network which can be used in the same way we used the ResNet network. There were many improvements made to the initial Inception block, and the current implementation available from PyTorch is Inception v3. Let's look at how we can use the Inception v3 model from `torchvision` to calculate pre-computed features. We will not go through the data loading process as we will be using the same data loaders from the previous ResNet section. We will look at the following important topics:

- Creating an Inception model
- Extracting convolutional features using `register_forward_hook`
- Creating a new dataset for the convoluted features

- Creating a fully connected model
- Training and validating the model

Creating an Inception model

The Inception v3 model has two branches, each of which generates an output, and in the original model training, we would merge the losses as we did for style transfer. As of now we are interested in using only one branch to calculate pre-convoluted features using Inception. Getting into the details of this is outside the scope of the book. If you are interested in knowing more about how it works, then going through the paper and the source code (`https://github.com/pytorch/vision/blob/master/torchvision/models/inception.py`) of the Inception model would help. We can disable one of the branches by setting the `aux_logits` parameter to `False`. The following code explains how to create a model and set the `aux_logits` parameter to `False`:

```
my_inception = inception_v3(pretrained=True)
my_inception.aux_logits = False
if is_cuda:
    my_inception = my_inception.cuda()
```

Extracting the convolution features from the Inception model is not straightforward, as with ResNet, so we will use the `register_forward_hook` to extract the activations.

Extracting convolutional features using register_forward_hook

We will be using the same techniques that we used to calculate activations for style transfer. The following is the `LayerActivations` class with some minor modifications, as we are interested in extracting only outputs of a particular layer:

```
class LayerActivations():
    features=[]

    def __init__(self,model):
        self.features = []
        self.hook = model.register_forward_hook(self.hook_fn)

    def hook_fn(self,module,input,output):

        self.features.extend(output.view(output.size(0),-1).cpu().data)

    def remove(self):

        self.hook.remove()
```

Apart from the `hook` function, the rest of the code is similar to what we have used for style transfer. As we are capturing the outputs of all the images and storing them, we will not be able to hold the data on **graphics processing unit** (**GPU**) memory. So we extract the tensors from GPU to CPU and just store the tensors instead of `Variable`. We are converting it back to tensors as the data loaders will work only with tensors. In the following code, we use the objects of

`LayerActivations` to extract the output of the Inception model at the last layer, excluding the average pooling layer, dropout and the linear layer. We are skipping the average pooling layer to avoid losing useful information in the data:

```
# Create LayerActivations object to store the output of inception model at a
trn_features = LayerActivations(my_inception.Mixed_7c)
trn_labels = []

# Passing all the data through the model , as a side effect the outputs will
# in the features list of the LayerActivations object.
for da,la in train_loader:
    _ = my_inception(Variable(da.cuda()))
    trn_labels.extend(la)
trn_features.remove()

# Repeat the same process for validation dataset .

val_features = LayerActivations(my_inception.Mixed_7c)
val_labels = []
for da,la in val_loader:
    _ = my_inception(Variable(da.cuda()))
    val_labels.extend(la)
val_features.remove()
```

Let's create the datasets and loaders required for the new convoluted features.

Creating a new dataset for the convoluted features

We can use the same `FeaturesDataset` class to create the new dataset and data loaders. In the following code, we create the datasets and the loaders:

```
#Dataset for pre computed features for train and validation data sets

trn_feat_dset = FeaturesDataset(trn_features.features,trn_labels)
val_feat_dset = FeaturesDataset(val_features.features,val_labels)

#Data loaders for pre computed features for train and validation data sets

trn_feat_loader = DataLoader(trn_feat_dset,batch_size=64,shuffle=True)
val_feat_loader = DataLoader(val_feat_dset,batch_size=64)
```

Let's create a new model to train on the pre-convoluted features.

Creating a fully connected model

A simple model may end in overfitting, so let's include dropout in the model. Dropout will help avoid overfitting. In the following code, we are creating our model:

```
class FullyConnectedModel(nn.Module):

    def __init__(self,in_size,out_size,training=True):
        super().__init__()
        self.fc = nn.Linear(in_size,out_size)

    def forward(self,inp):
        out = F.dropout(inp, training=self.training)
        out = self.fc(out)
        return out

# The size of the output from the selected convolution feature
fc_in_size = 131072

fc = FullyConnectedModel(fc_in_size,classes)
if is_cuda:
    fc = fc.cuda()
```

Once the model is created, we can train the model.

Training and validating the model

We use the same fit and training logic as seen in the previous ResNet and other examples. We will just look at the training code and the results from it:

```
for epoch in range(1,10):
    epoch_loss, epoch_accuracy = fit(epoch,fc,trn_feat_loader,phase='training
    val_epoch_loss , val_epoch_accuracy = fit(epoch,fc,val_feat_loader,phase=
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    val_losses.append(val_epoch_loss)
    val_accuracy.append(val_epoch_accuracy)

#Results

training loss is 0.78 and training accuracy is 22825/23000 99.24
validation loss is 5.3 and validation accuracy is 1947/2000 97.35
training loss is 0.84 and training accuracy is 22829/23000 99.26
validation loss is 5.1 and validation accuracy is 1952/2000 97.6
training loss is 0.69 and training accuracy is 22843/23000 99.32
validation loss is 5.1 and validation accuracy is 1951/2000 97.55
training loss is 0.58 and training accuracy is 22852/23000 99.36
validation loss is 4.9 and validation accuracy is 1953/2000 97.65
training loss is 0.67 and training accuracy is 22862/23000 99.4
validation loss is 4.9 and validation accuracy is 1955/2000 97.75
training loss is 0.54 and training accuracy is 22870/23000 99.43
validation loss is 4.8 and validation accuracy is 1953/2000 97.65
training loss is 0.56 and training accuracy is 22856/23000 99.37
validation loss is 4.8 and validation accuracy is 1955/2000 97.75
training loss is 0.7 and training accuracy is 22841/23000 99.31
validation loss is 4.8 and validation accuracy is 1956/2000 97.8
training loss is 0.47 and training accuracy is 22880/23000 99.48
validation loss is 4.7 and validation accuracy is 1956/2000 97.8
```

Looking at the results, the Inception model achieves 99% accuracy on the training and 97.8% accuracy on the validation dataset. As we are pre-computing and holding all the features in the memory, it takes less than a few minutes to train the models. If you are running out of memory when you run the program on your machine, then you may need to avoid holding the features in the memory.

We will look at another interesting architecture, DenseNet, which has become very popular in the last year.

Densely connected convolutional networks – DenseNet

Some of the successful and popular architectures, such as ResNet and Inception, have shown the importance of deeper and wider networks. ResNet uses shortcut connections to build deeper networks. DenseNet takes it to a new level by introducing connections from each layer to all other subsequent layers, that is a layer where one could receive all the feature maps from the previous layers. Symbolically, it would look like the following:

$$X_l = H_l(x_0, x_1, x_2, \ldots, x_{l-1})$$

The following figure describes what a five-layer dense block would look like:

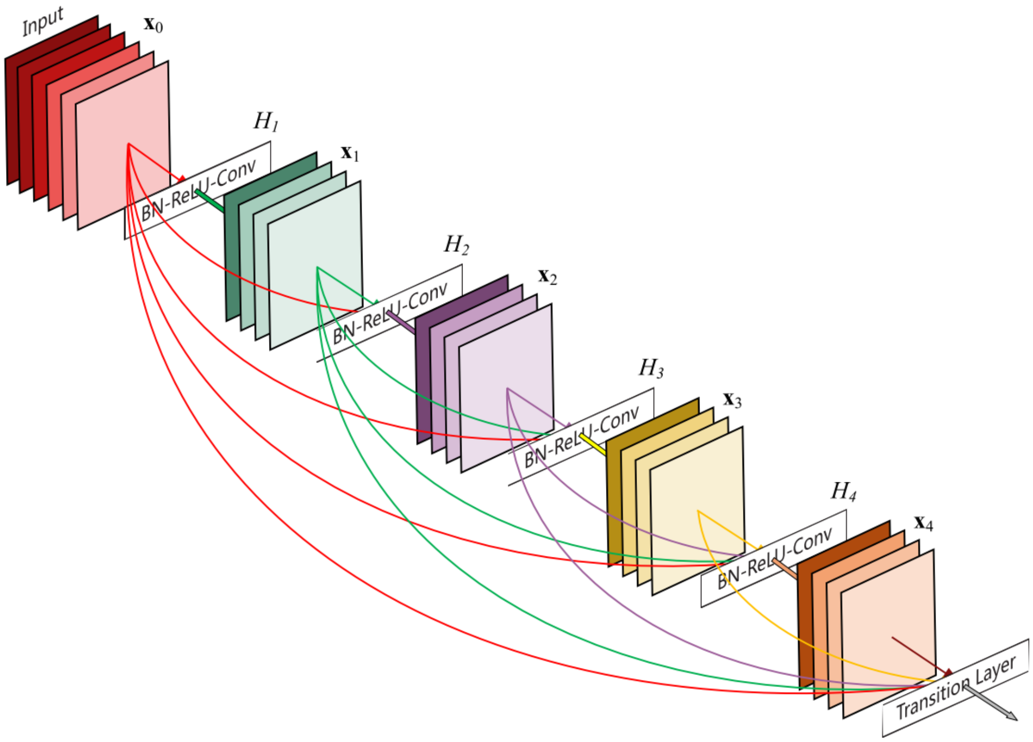

Input
$\mathbf{x}_0$
H_1
BN-ReLU-Conv
$\mathbf{x}_1$
H_2
BN-ReLU-Conv
$\mathbf{x}_2$
H_3
BN-ReLU-Conv
$\mathbf{x}_3$
H_4
BN-ReLU-Conv
$\mathbf{x}_4$
Transition Layer

Image source: https://arxiv.org/abs/1608.06993

There is a DenseNet implementation of torchvision (`https://github.com/pytorch/vision/blob/master/torchvision/models/densenet.py`). Let's look at two major functionalities, `_DenseBlock` and `_DenseLayer`.

DenseBlock

Let's look at the code for `DenseBlock` and then walk through it:

```
class _DenseBlock(nn.Sequential):
    def __init__(self, num_layers, num_input_features, bn_size, growth_rate,
        super(_DenseBlock, self).__init__()
        for i in range(num_layers):
            layer = _DenseLayer(num_input_features + i * growth_rate, growth_
            self.add_module('denselayer%d' % (i + 1), layer)
```

`DenseBlock` is a sequential module where we add layers in a sequential order. Based on the number of layers (`num_layers`) in the block, we add that number of `_Denselayer` objects along with a name to it. All the magic is happening inside the `DenseLayer`. Let's look at what goes on inside the `DenseLayer`.

DenseLayer

One good way to learn how a particular network works is to look at the source code. PyTorch has a very clean implementation and most of the time is easily readable. Let's look at the `DenseLayer` implementation:

```
class _DenseLayer(nn.Sequential):
    def __init__(self, num_input_features, growth_rate, bn_size, drop_rate):
        super(_DenseLayer, self).__init__()
        self.add_module('norm.1', nn.BatchNorm2d(num_input_features)),
        self.add_module('relu.1', nn.ReLU(inplace=True)),
        self.add_module('conv.1', nn.Conv2d(num_input_features, bn_size *
                        growth_rate, kernel_size=1, stride=1, bias=False)),
        self.add_module('norm.2', nn.BatchNorm2d(bn_size * growth_rate)),
        self.add_module('relu.2', nn.ReLU(inplace=True)),
        self.add_module('conv.2', nn.Conv2d(bn_size * growth_rate, growth_rat
                        kernel_size=3, stride=1, padding=1, bias=False)),
        self.drop_rate = drop_rate

    def forward(self, x):
        new_features = super(_DenseLayer, self).forward(x)
        if self.drop_rate > 0:
            new_features = F.dropout(new_features, p=self.drop_rate, training
        return torch.cat([x, new_features], 1)
```

If you are new to inheritance in Python, then the preceding code may not look intuitive. The `_DenseLayer` is a subclass of `nn.Sequential`; let's look at what goes on inside each method.

In the `__init__` method, we add all the layers that the input data needs to be passed to. It is quite similar to all the other network architectures we have seen.

The magic happens in the `forward` method. We pass the input to the `forward` method of the `super` class, which is `nn.Sequential`. Let's look at what happens in the `forward` method of the sequential class (`https://github.com/pytorch/pytorch/blob/409b1c8319ecde4bd62fcf98d0a6658ae7a4ab23/torch/nn/modules/container.py`):

```
def forward(self, input):
    for module in self._modules.values():
        input = module(input)
    return input
```

The input is passed through all the layers that were previously added to the sequential block and the output is concatenated to the input. The process is repeated for the required number of layers in a block.

With the understanding of how a `DenseNet` block works, let's explore how we can use DenseNet for calculating pre-convoluted features and building a classifier model on top of it. At a high level, the DenseNet implementation is similar to the VGG implementation. The DenseNet implementation also has a features module, which contains all the dense blocks, and a classifier module, which contains the fully connected model. We will be going through the following steps to build the model. We will be skipping most of the part that is similar to what we have seen for Inception and ResNet, such as creating the data loader and datasets. Also, we will discuss the following steps in detail:

- Creating a DenseNet model
- Extracting DenseNet features
- Creating a dataset and loaders
- Creating a fully connected model and train

By now, most of the code will be self-explanatory.

Creating a DenseNet model

Torchvision has a pretrained DenseNet model with different layer options (121, 169, 201, 161). We have chosen the model with `121` layers. As discussed, the DenseNet has two modules: features (containing the dense blocks), and classifier (fully connected block). As we are using DenseNet as an image feature extractor, we will only use the feature module:

```
my_densenet = densenet121(pretrained=True).features
if is_cuda:
    my_densenet = my_densenet.cuda()

for p in my_densenet.parameters():
    p.requires_grad = False
```

Let's extract the DenseNet features from the images.

Extracting DenseNet features

It is quite similar to what we did for Inception, except we are not using `register_forward_hook` to extract features. The following code shows how the DenseNet features are extracted:

```
#For training data
trn_labels = []
trn_features = []

#code to store densenet features for train dataset.
for d,la in train_loader:
    o = my_densenet(Variable(d.cuda()))
    o = o.view(o.size(0),-1)
    trn_labels.extend(la)
    trn_features.extend(o.cpu().data)

#For validation data
val_labels = []
val_features = []

#Code to store densenet features for validation dataset.
for d,la in val_loader:
    o = my_densenet(Variable(d.cuda()))
    o = o.view(o.size(0),-1)
    val_labels.extend(la)
    val_features.extend(o.cpu().data)
```

The preceding code is similar to what we have seen for Inception and ResNet.

Creating a dataset and loaders

We will use the same `FeaturesDataset` class that we created for ResNet and use it to create data loaders for the `train` and `validation` dataset in the following code:

```
# Create dataset for train and validation convolution features
trn_feat_dset = FeaturesDataset(trn_features,trn_labels)
val_feat_dset = FeaturesDataset(val_features,val_labels)

# Create data loaders for batching the train and validation datasets
trn_feat_loader = DataLoader(trn_feat_dset,batch_size=64,shuffle=True,drop_la
val_feat_loader = DataLoader(val_feat_dset,batch_size=64)
```

Time to create the model and train it.

Creating a fully connected model and train

We will use a simple linear model, similar to what we used in ResNet and Inception. The following code shows the network architecture which we will be using to train the model:

```
class FullyConnectedModel(nn.Module):

    def __init__(self,in_size,out_size):
        super().__init__()
        self.fc = nn.Linear(in_size,out_size)

    def forward(self,inp):
        out = self.fc(inp)
        return out

fc = FullyConnectedModel(fc_in_size,classes)
if is_cuda:
    fc = fc.cuda()
```

We will use the same `fit` method to train the preceding model. The following code snippet shows the training code, along with the results:

```
train_losses , train_accuracy = [],[]
val_losses , val_accuracy = [],[]
for epoch in range(1,10):
    epoch_loss, epoch_accuracy = fit(epoch,fc,trn_feat_loader,phase='training
    val_epoch_loss , val_epoch_accuracy = fit(epoch,fc,val_feat_loader,phase=
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    val_losses.append(val_epoch_loss)
    val_accuracy.append(val_epoch_accuracy)
```

The result of the preceding code is:

```
# Results

training loss is 0.057 and training accuracy is 22506/23000 97.85
validation loss is 0.034 and validation accuracy is 1978/2000 98.9
training loss is 0.0059 and training accuracy is 22953/23000 99.8
validation loss is 0.028 and validation accuracy is 1981/2000 99.05
training loss is 0.0016 and training accuracy is 22974/23000 99.89
validation loss is 0.022 and validation accuracy is 1983/2000 99.15
training loss is 0.00064 and training accuracy is 22976/23000 99.9
validation loss is 0.023 and validation accuracy is 1983/2000 99.15
training loss is 0.00043 and training accuracy is 22976/23000 99.9
validation loss is 0.024 and validation accuracy is 1983/2000 99.15
training loss is 0.00033 and training accuracy is 22976/23000 99.9
validation loss is 0.024 and validation accuracy is 1984/2000 99.2
training loss is 0.00025 and training accuracy is 22976/23000 99.9
validation loss is 0.024 and validation accuracy is 1984/2000 99.2
training loss is 0.0002 and training accuracy is 22976/23000 99.9
validation loss is 0.025 and validation accuracy is 1985/2000 99.25
training loss is 0.00016 and training accuracy is 22976/23000 99.9
validation loss is 0.024 and validation accuracy is 1986/2000 99.3
```

The preceding algorithm was able to achieve a maximum training accuracy of 99%, and 99% validation accuracy. Your results could change as the `validation` dataset you create may have different images.

Some of the advantages of DenseNet are:

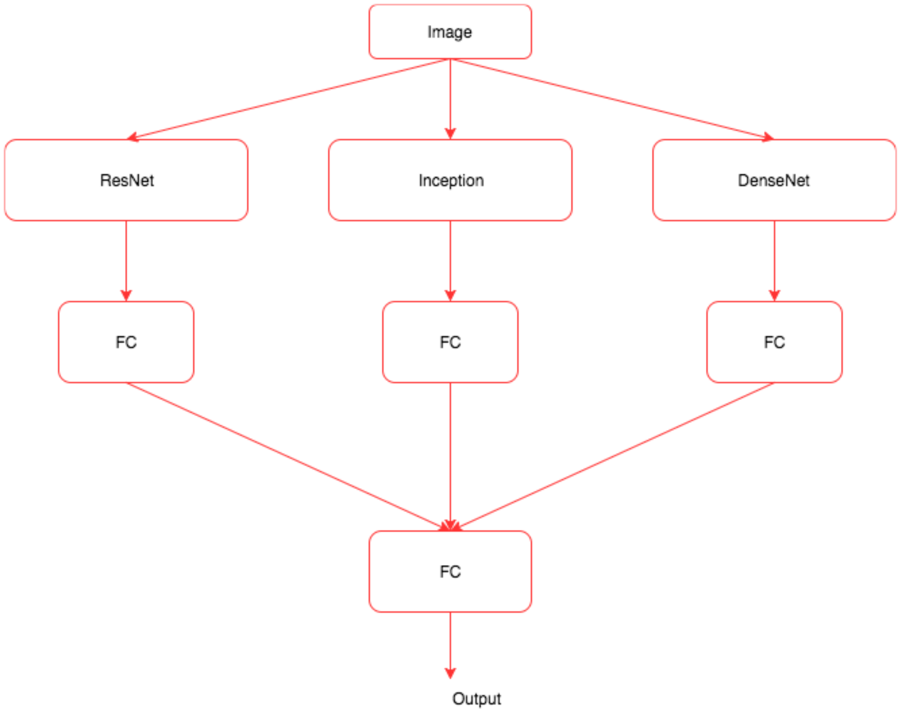
- It substantially reduces the number of parameters required
- It alleviates the vanishing gradient problem
- It encourages feature reuse

In this next section, we will explore how we can build a model that combines the advantage of the convoluted features computed, using the different models of ResNet, Inception, and DenseNet.

Model ensembling

There could be times when we would need to try to combine multiple models to build a very powerful model. There are many techniques that can be used for building an ensemble model. In this section, we will learn how to combine outputs using the features generated by three different models (ResNet, Inception, and DenseNet) to build a powerful model. We will be using the same dataset that we used for other examples in this chapter.

The architecture for the ensemble model would look like this:



This image shows what we are going to do in the ensemble model, which can be summarized in the following steps:

1. Create three models
2. Extract the image features using the created models
3. Create a custom dataset which returns features of all the three models along with the labels
4. Create model similar to the architecture in the preceding figure
5. Train and validate the model

Let's explore each of the steps in detail.

Creating models

Let's create all the three required models, as shown in the following code:

```
#Create ResNet model
my_resnet = resnet34(pretrained=True)

if is_cuda:
    my_resnet = my_resnet.cuda()

my_resnet = nn.Sequential(*list(my_resnet.children())[:-1])

for p in my_resnet.parameters():
    p.requires_grad = False

#Create inception model

my_inception = inception_v3(pretrained=True)
my_inception.aux_logits = False
if is_cuda:
    my_inception = my_inception.cuda()
for p in my_inception.parameters():
    p.requires_grad = False

#Create densenet model

my_densenet = densenet121(pretrained=True).features
if is_cuda:
    my_densenet = my_densenet.cuda()

for p in my_densenet.parameters():
    p.requires_grad = False
```

Now we have all the models, let's extract the features from the images.

Extracting the image features

Here, we combine all the logic that we have seen individually for the algorithms in the chapter:

```
### For ResNet

trn_labels = []
trn_resnet_features = []
for d,la in train_loader:
    o = my_resnet(Variable(d.cuda()))
    o = o.view(o.size(0),-1)
    trn_labels.extend(la)
    trn_resnet_features.extend(o.cpu().data)
val_labels = []
val_resnet_features = []
for d,la in val_loader:
    o = my_resnet(Variable(d.cuda()))
    o = o.view(o.size(0),-1)
    val_labels.extend(la)
    val_resnet_features.extend(o.cpu().data)

### For Inception

trn_inception_features = LayerActivations(my_inception.Mixed_7c)
for da,la in train_loader:
    _ = my_inception(Variable(da.cuda()))

trn_inception_features.remove()

val_inception_features = LayerActivations(my_inception.Mixed_7c)
for da,la in val_loader:
    _ = my_inception(Variable(da.cuda()))

val_inception_features.remove()

### For DenseNet
```

```
trn_densenet_features = []
for d,la in train_loader:
    o = my_densenet(Variable(d.cuda()))
    o = o.view(o.size(0),-1)

    trn_densenet_features.extend(o.cpu().data)


val_densenet_features = []
for d,la in val_loader:
    o = my_densenet(Variable(d.cuda()))
    o = o.view(o.size(0),-1)
    val_densenet_features.extend(o.cpu().data)
```

By now, we have created image features using all the models. If you are facing memory issues, then you can either remove one of the models, or stop storing the features in the memory, which could be slow to train. If you are running this on a CUDA instance, then you can go for a more powerful instance.

Creating a custom dataset along with data loaders

We will not be able to use the `FeaturesDataset` class as it is, since it was developed to pick from the output of only one model. So, the following implementation contains minor changes to the `FeaturesDataset` class to accommodate all the three different generated features:

```
class FeaturesDataset(Dataset):

    def __init__(self,featlst1,featlst2,featlst3,labellst):
        self.featlst1 = featlst1
        self.featlst2 = featlst2
        self.featlst3 = featlst3
        self.labellst = labellst

    def __getitem__(self,index):
        return (self.featlst1[index],self.featlst2[index],self.featlst3[index

    def __len__(self):
        return len(self.labellst)

trn_feat_dset = FeaturesDataset(trn_resnet_features,trn_inception_features.fe
val_feat_dset = FeaturesDataset(val_resnet_features,val_inception_features.fe
```

We have made changes to the `__init__` method to store all the features generated from different models, and the `__getitem__` method to retrieve the features and label of an image. Using the `FeatureDataset` class, we created dataset instances for both training and validation data. Once the dataset is created, we can use the same data loader for batching data, as shown in the following code:

```
trn_feat_loader = DataLoader(trn_feat_dset,batch_size=64,shuffle=True)
val_feat_loader = DataLoader(val_feat_dset,batch_size=64)
```

Creating an ensembling model

We need to create a model that is similar to the architecture diagram show previously. The following code implements this:

```
class EnsembleModel(nn.Module):

    def __init__(self,out_size,training=True):
        super().__init__()
        self.fc1 = nn.Linear(8192,512)
        self.fc2 = nn.Linear(131072,512)
        self.fc3 = nn.Linear(82944,512)
        self.fc4 = nn.Linear(512,out_size)

    def forward(self,inp1,inp2,inp3):
        out1 = self.fc1(F.dropout(inp1,training=self.training))
        out2 = self.fc2(F.dropout(inp2,training=self.training))
        out3 = self.fc3(F.dropout(inp3,training=self.training))
        out = out1 + out2 + out3
        out = self.fc4(F.dropout(out,training=self.training))
        return out

em = EnsembleModel(2)
if is_cuda:
    em = em.cuda()
```

In the preceding code, we create three linear layers that take features generated from different models. We sum up all the outputs from these three linear layers and pass them on to another linear layer, which maps them to the required categories. To prevent the model from overfitting, we have used dropouts.

Training and validating the model

We need to make some minor changes to the `fit` method to accommodate the three input-values generated from the data loader. The following code implements the new `fit` function:

```
def fit(epoch,model,data_loader,phase='training',volatile=False):
    if phase == 'training':
        model.train()
    if phase == 'validation':
        model.eval()
        volatile=True
    running_loss = 0.0
    running_correct = 0
    for batch_idx , (data1,data2,data3,target) in enumerate(data_loader):
        if is_cuda:
            data1,data2,data3,target = data1.cuda(),data2.cuda(),data3.cuda()
        data1,data2,data3,target = Variable(data1,volatile),Variable(data2,v
        if phase == 'training':
            optimizer.zero_grad()
        output = model(data1,data2,data3)
        loss = F.cross_entropy(output,target)

        running_loss += F.cross_entropy(output,target,size_average=False).dat
        preds = output.data.max(dim=1,keepdim=True)[1]
        running_correct += preds.eq(target.data.view_as(preds)).cpu().sum()
        if phase == 'training':
            loss.backward()
            optimizer.step()

    loss = running_loss/len(data_loader.dataset)
    accuracy = 100. * running_correct/len(data_loader.dataset)

    print(f'{phase} loss is {loss:{5}.{2}} and {phase} accuracy is {running_c
    return loss,accuracy
```

As you can see from the previous code, most of it remains the same, except that the loader returns three inputs and one label. So, we make changes in the function, which is self-explanatory.

The following code shows the training code:

```
train_losses , train_accuracy = [],[]
val_losses , val_accuracy = [],[]
for epoch in range(1,10):
    epoch_loss, epoch_accuracy = fit(epoch,em,trn_feat_loader,phase='training
    val_epoch_loss , val_epoch_accuracy = fit(epoch,em,val_feat_loader,phase=
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    val_losses.append(val_epoch_loss)
    val_accuracy.append(val_epoch_accuracy)
```

The result of the preceding code is as follows:

```
#Results

training loss is 7.2e+01 and training accuracy is 21359/23000 92.87
validation loss is 6.5e+01 and validation accuracy is 1968/2000 98.4
training loss is 9.4e+01 and training accuracy is 22539/23000 98.0
validation loss is 1.1e+02 and validation accuracy is 1980/2000 99.0
training loss is 1e+02 and training accuracy is 22714/23000 98.76
validation loss is 1.4e+02 and validation accuracy is 1976/2000 98.8
training loss is 7.3e+01 and training accuracy is 22825/23000 99.24
validation loss is 1.6e+02 and validation accuracy is 1979/2000 98.95
training loss is 7.2e+01 and training accuracy is 22845/23000 99.33
validation loss is 2e+02 and validation accuracy is 1984/2000 99.2
training loss is 1.1e+02 and training accuracy is 22862/23000 99.4
validation loss is 4.1e+02 and validation accuracy is 1975/2000 98.75
training loss is 1.3e+02 and training accuracy is 22851/23000 99.35
validation loss is 4.2e+02 and validation accuracy is 1981/2000 99.05
training loss is 2e+02 and training accuracy is 22845/23000 99.33
validation loss is 6.1e+02 and validation accuracy is 1982/2000 99.1
training loss is 1e+02 and training accuracy is 22917/23000 99.64
validation loss is 5.3e+02 and validation accuracy is 1986/2000 99.3
```

The ensemble model achieves a 99.6% training accuracy and a validation accuracy of 99.3%. Though ensemble models are powerful, they are computationally expensive. They are good techniques to use when you are solving problems in competitions such as Kaggle.

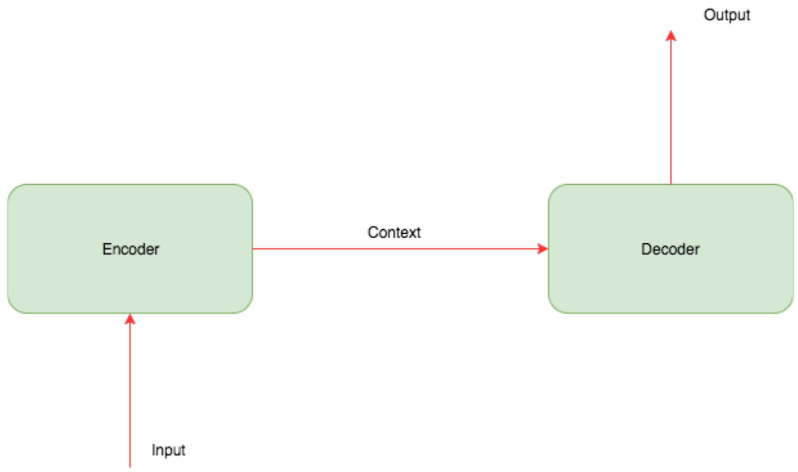
Encoder-decoder architecture

Almost all the deep learning algorithms we have seen in the book are good at learning how to map training data to their corresponding labels. We cannot use them directly for tasks where the model needs to learn from a sequence and generate another sequence or an image. Some of the example applications are:

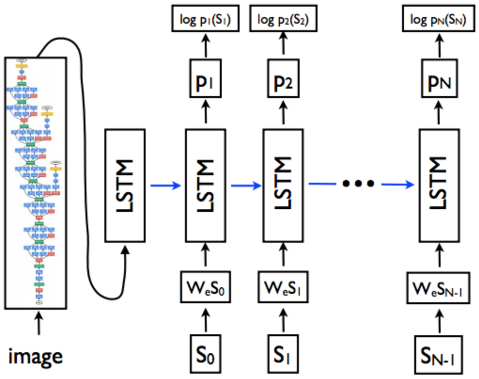
- Language translation
- Image captioning
- Image generation (seq2img)
- Speech recognition
- Question answering

Most of these problems can be seen as some form of sequence-to-sequence mapping, and these can be solved using a family of architectures called **encoder–decoder architectures**. In this section, we will learn about the intuition behind these architectures. We will not be looking at the implementation of these networks, as they need to be studied in more detail.

At a high level, an encoder–decoder architecture would look like the following:


Output
Encoder
Context
Decoder
Input

An encoder is usually a **recurrent neural network** (**RNN**) (for sequential data) or a **Convolution Neural Network** (**CNN**) (for images) that takes in an image or a sequence and converts it into a fixed length vector which encodes all the information. The decoder is another RNN or CNN, which learns to decode the vector generated by the encoder and generates a new sequence of data. The following image shows how the encoder–decoder architecture looks for an image captioning system:


$\log p_1(S_1)$
$\log p_2(S_2)$
$\log p_N(S_N)$
p_1
p_2
p_N
LSTM
LSTM
LSTM
LSTM
W_eS_0
W_eS_1
W_eS_{N-1}
S_0
S_1
S_{N-1}
image

Encoder-decoder architecture for image captioning system
Image source: https://arxiv.org/pdf/1411.4555.pdf

Let's look in more detail at what happens inside an encoder and a decoder architecture for an image captioning system.

Encoder

For an image captioning system, we will preferably use a trained architecture, such as ResNet or Inception, for extracting features from the image. Like we did for the ensemble model, we can output a fixed vector length by using a linear layer, and then make that linear layer trainable.

Decoder

Decoder is a **Long Short-Term Memory** (**LSTM**) layer which will generate a caption for an image. To build a simple model, we can just pass the encoder embedding as input to the LSTM only once. But it could be quite challenging for the decoder to learn; instead, it is common practice to provide the encoder embedding at every step of the decoder. Intuitively, a decoder learns to generate a sequence of text that best describes the caption of a given image.

Summary

In this chapter, we explored some modern architectures, such as ResNet, Inception, and DenseNet. We also explored how we can use these models for transfer learning and ensembling, and introduced the encoder–decoder architecture, which powers a lot of systems, such as language translation systems.

In the next chapter, we will arrive at a conclusion of what we have achieved in our learning journey through the book, as well as discuss where can you go from here. We will visit a plethora of resources on PyTorch and some cool deep learning projects that have been created or are undergoing research using PyTorch.

What Next?

You made it! Thanks for reading *Deep Learning with PyTorch*. You should have a firm understanding of the core mechanisms and the **application program interface** (**API**) required for building deep learning applications using PyTorch. By now, you should be comfortable in using all the fundamental blocks that power most of the modern-day deep learning algorithms.

What next?

In this chapter, we will summarize what we learned in this book and further explore different projects and resources. These projects and resources will help you further in the journey of keeping yourself up-to-date with the latest research.

Overview

This section provides a bird's-eye view of what we learned across the book:

- History of **artificial intelligence** (**AI**), machine learning—how various improvements in hardware and algorithms triggered huge successes in the implementation of deep learning across different applications.
- How to use various building blocks of PyTorch, such as variables, tensors, and `nn.module`, to develop neural networks.
- Understanding the different processes involved in training a neural network, such as the PyTorch dataset for data preparation, data loaders for batching tensors, the `torch.nn` package for creating network architectures, and using PyTorch loss functions and optimizers.
- We saw different types of machine learning problems along with challenges, such as overfitting and underfitting. We also went through different techniques, such as data augmentation, adding dropouts, and using batch normalization to prevent overfitting.
- We learned the different building blocks of **Convolution Neural Networks** (**CNNs**), and also learned about transfer learning, which helps us to use a pretrained model. We also

saw techniques, such as using pre-convoluted features, which helps in reducing the time taken to train the models.

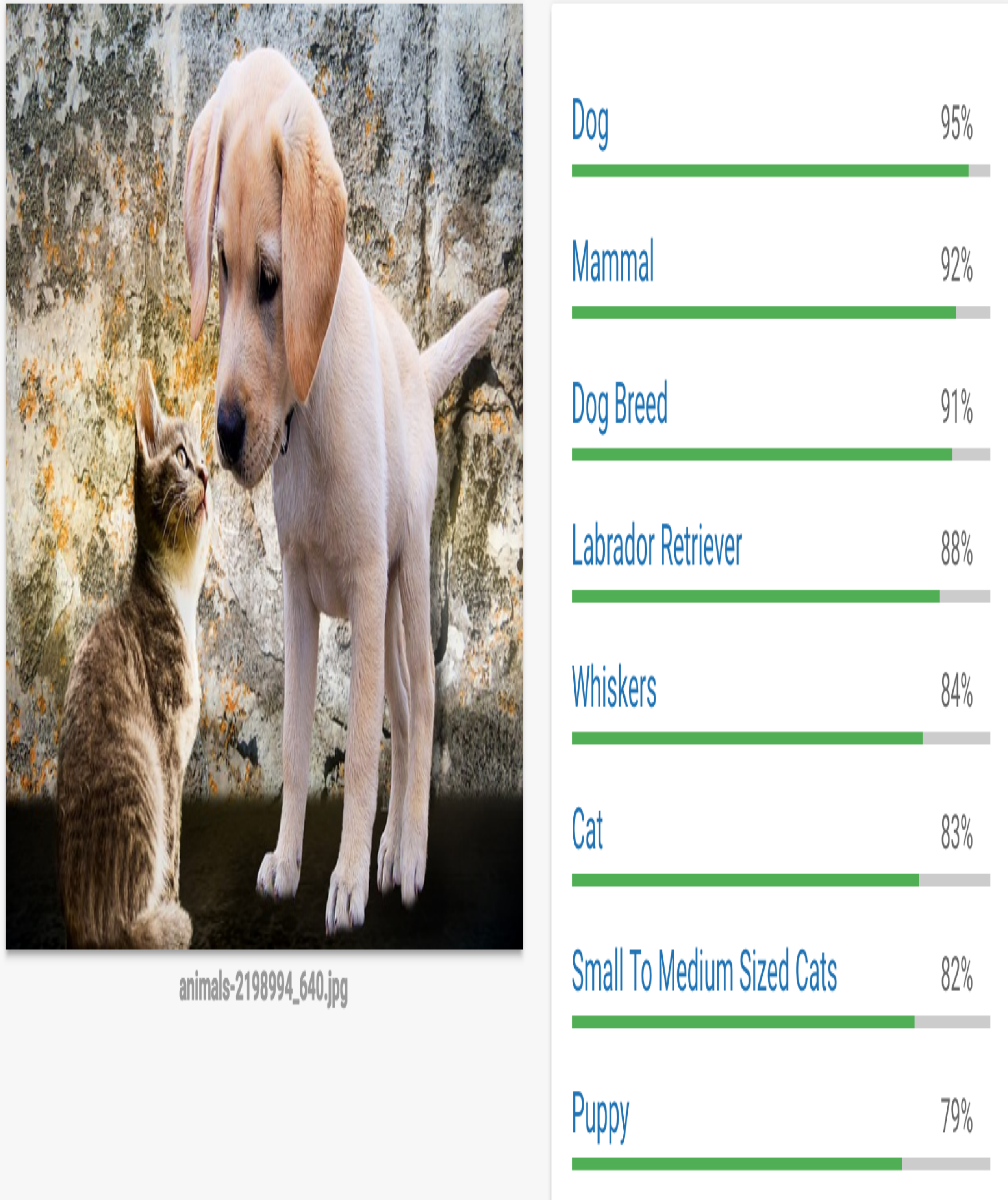
- We learned about word embeddings and how to use them for text classification problems. We also explored how we can use pretrained word embedding. We explored **recurrent neural network** (**RNN**), its variants such as **Long Short-Term Memory** (**LSTM**), and how to use them for text classification problems.
- We explored generative models and learned how PyTorch can be used for creating artistic style transfer, and for creating new CIFAR images using a **generative adversarial network** (**GAN**). We also explored language modeling techniques which can be used to generate new text or to create domain-specific embedding.
- We explored modern architectures, such as ResNet, Inception, DenseNet and encode-decoder architecture. We also saw how these models can be used for transfer learning. We also built an ensemble model by combining all these models.

Interesting ideas to explore

Most of the concepts that we learned in the book form the foundation of modern applications that are powered by deep learning. In this section, we will look at the different interesting projects that we can do that are related to computer vision and **natural language processing** (**NLP**).

Object detection

All the examples we have seen in this book help you in detecting whether a given image is this (cat) or that (dog). But, to solve some of the problems in the real world, you may need to identify different objects in an image, such as shown here:



```
Dog                          95%
Mammal                       92%
Dog Breed                    91%
Labrador Retriever           88%
Whiskers                     84%
Cat                          83%
Small To Medium Sized Cats   82%
Puppy                        79%
```

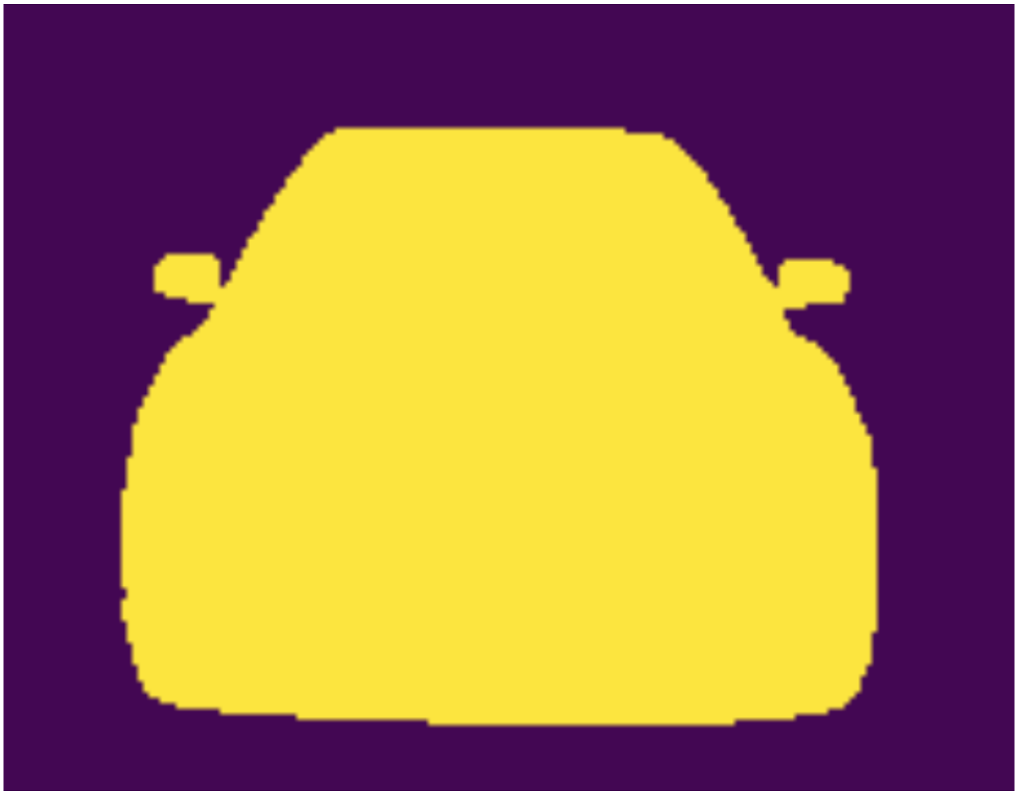
Output of object detection algorithm

This image shows the output of an object detection algorithm where the algorithm is detecting objects such as a beautiful dog and cat . Just as there are off-the-shelf algorithms for image classification, there are a bunch of amazing algorithms that can help in building object recognition systems. Here is a list of some of the important algorithms and the papers that mention them:

- **Single Shot Multibox Detector (SSD)** `https://arxiv.org/abs/1512.02325`
- Faster RCNN `https://arxiv.org/abs/1506.01497`
- YOLO2 `https://arxiv.org/abs/1612.08242`

Image segmentation

Let's assume you are reading this book from the terrace of a building. What do you see around you? Can you draw an outline of what you see? If you are a good artist, unlike me, then you would have probably drawn a couple of buildings, trees, birds, and a few more interesting things surrounding you. Image segmentation algorithms try to capture something similar. Given an image, they generate a prediction for each pixel, identifying which class each pixel belongs to. The following image shows what image segmentation algorithms identify:



Output of image segmentation algorithm

Some of the important algorithms that you may want to explore for image segmentation are:

- R-CNN `https://arxiv.org/abs/1311.2524`
- Fast R-CNN `https://arxiv.org/abs/1504.08083`
- Faster R-CNN `https://arxiv.org/abs/1506.01497`
- Mask R-CNN `https://arxiv.org/abs/1703.06870`

OpenNMT in PyTorch

The **Open-Source Neural Machine Translation** (**OpenNMT**) (`https://github.com/OpenNMT/OpenNMT-py`) project helps in building a lot of applications that are powered by the encoder-decoder architecture. Some of the applications that you can build are translation systems, text summarization, and image-to-text.

Alien NLP

Alien NLP is an open source project built on PyTorch which enables us to do many NLP tasks much more easily. There is a demo page (`http://demo.allennlp.org/machine-comprehension`) that you should look at to understand what you can build using Alien NLP.

fast.ai – making neural nets uncool again

One of my favorite places to learn about deep learning, and a great place of inspiration, is a MOOC with the sole motive of making deep learning accessible to all, organized by two amazing mentors from *fast.ai* (`http://www.fast.ai/`), Jeremy Howard and Rachel Thomas. For a new version of their course, they built an incredible framework (`https://github.com/fastai/fastai`) on top of PyTorch, making it much easier and quicker to build applications. If you have not already started their course, I would strongly recommend you start it. Exploring how the *fast.ai* framework is built will give you great insight into many powerful techniques.

Open Neural Network Exchange

Open Neural Network Exchange (**ONNX**) (`http://onnx.ai/`) is the first step towards an open ecosystem that empowers you to choose the right tools as the project evolves. ONNX provides an open source format for deep learning models. It defines an extensible computation graph model, as well as definitions of built-in operators and standard data types. Caffe2, PyTorch, Microsoft Cognitive Toolkit, Apache MXNet, and other tools are developing ONNX support. This project can help in product-ionizing PyTorch models.

How to keep yourself updated

Social media platforms, particularly Twitter, help you to stay updated in the field. There are many people you can follow. If you are unsure of where to start, I would recommend following Jeremy Howard (`https://twitter.com/jeremyphoward`), and any interesting people he may follow. By doing this, you would be forcing the Twitter recommendation system to work for you.

Another important Twitter account you need to follow is PyTorch's (`https://twitter.com/PyTorch`). The amazing people behind PyTorch have some great content being shared.

If you are looking for research papers, then look at *arxiv-sanity* (`http://www.arxiv-sanity.com/`), where many smart researchers publish their papers.

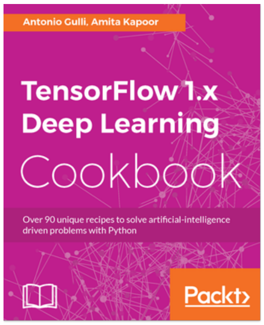
More great resources for learning about PyTorch are its tutorials (`http://pytorch.org/tutorials/`), its source code (`https://github.com/pytorch/pytorch`), and its documentation (`http://pytorch.org/docs/0.3.0/`).

Summary

There is much more to deep learning and PyTorch. PyTorch is a relatively new framework, which, at the time of writing this chapter, is a year old. There is much more to learn and explore, so happy learning. All the best.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

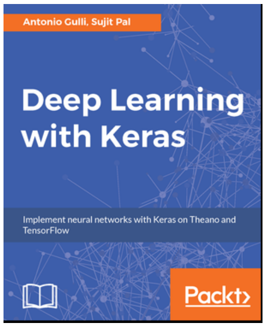


TensorFlow 1.x Deep Learning Cookbook
Antonio Gulli, Amita Kapoor

ISBN: 978-1-78829-359-4

- Install TensorFlow and use it for CPU and GPU operations.
- Implement DNNs and apply them to solve different AI-driven problems.
- Leverage different data sets such as MNIST, CIFAR-10, and Youtube8m with TensorFlow and learn how to access and use them in your code.
- Use TensorBoard to understand neural network architectures, optimize the learning process, and peek inside the neural network black box.

- Use different regression techniques for prediction and classification problems
- Build single and multilayer perceptrons in TensorFlow
- Implement CNN and RNN in TensorFlow, and use it to solve real-world use cases.
- Learn how restricted Boltzmann Machines can be used to recommend movies.
- Understand the implementation of Autoencoders and deep belief networks, and use them for emotion detection.
- Master the different reinforcement learning methods to implement game playing agents.
- GANs and their implementation using TensorFlow.


Antonio Gulli, Sujit Pal
Deep Learning with Keras
Implement neural networks with Keras on Theano and TensorFlow
Packt

Deep Learning with Keras
Antonio Gulli, Sujit Pal

ISBN: 978-1-78712-842-2

- Optimize step-by-step functions on a large neural network using the Backpropagation Algorithm

- Fine-tune a neural network to improve the quality of results
- Use deep learning for image and audio processing
- Use Recursive Neural Tensor Networks (RNTNs) to outperform standard word embedding in special cases
- Identify problems for which Recurrent Neural Network (RNN) solutions are suitable
- Explore the process required to implement Autoencoders
- Evolve a deep neural network using reinforcement learning

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!